Basics of Software Testing

A course by Quality House Ltd.

Based on ISTQB® Certified Tester,
Foundation Level Syllabus 2011

Version 3.3 June 2015

General

Course Contents

About

This document presents the material needed for the preparation for certification as a software tester and contains the complete set of student notes for the Quality House course - Basics of Software Testing.

After having covered the material you will be expected to be able to demonstrate that you have a basic knowledge of software testing fundamental principles and terminology.

Prerequisites

In order to feel comfortable during the course and when studying the material, it is expected that you have some background in software development or software testing, such as:

- experience as a system or user acceptance tester or
- as a software developer.
- as a manager of software or website projects
- as a student of computer science, programming, or testing

You are also expected to understand written English since all course materials are in English.

You may find this material too difficult or incomprehensible if you are completely new to computers or if you have no previous background or experience with software development or testing.

If you are completely new to software testing, it is recommended that you start with a one-day overview of testing and then gain a few months' experience before attempting this qualification.

You should be aware that there is a lot of terminology and facts that need to be learned in order to pass the exam.

Abbreviations

Throughout this document you will encounter the following abbreviations:

BCS	British Computer Society
BVA	Boundary Value Analysis
CASE	Computer Aided Software Engineering
CAST	Computer Aided Software Testing
CI	Configuration Item
CM	Configuration Management
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CSF	Critical Success Factors
CUI	Character-based User Interface
EP	Equivalence Partitioning
GUI	Graphical User Interface
IIT	Illinois Institute of Technology
iSQI	International Software Quality Institute
ISTQB	International Software Testing Qualification Board
RBOC	Regional Bell Operating Company
SDLC	System Development Life Cycle *or* Software Development Life Cycle
SEETB	South East European Testing Board
SEI	Software Engineering Institute
SIG	Special Interest Group
SIGiST	Special Interest Group in Software Testing
SPICE	Software Process Improvement and Capability dEtermination
SW-CMM	Capability Maturity Model for Software
SUT	System Under Test
TMM	Testing Maturity Model
TPI	Test Process Improvement
WIMP	Windows, Icons, Menus, and P

Introduction

Certification

All problems in software seem to boil down to quality or actually the lack of it. Quality when analyzed leads to project management and to good practices and standards. The way to subscribe to them is to use some sort of process and to check at various points what you are doing and how you are doing it - thus testing.

If you were the manager of a company you would need to know that your testers are good at what they are doing and know what to do. One of the ways to assure this is to work with people who have been certified.

If you have the time and resources you could check the skills of your testers when you hire them - then you do not need certification. If there were software testing programs at universities, it would have been easier.

One can have reasonable confidence that certified testers can be expected to

- Analyze the project and evaluate the risks to the business to design and plan the project inspections and tests.
- Define appropriate test targets, techniques, tasks, and schedules based on their awareness of the limitations with which they work: framework, technology, process, people, risks, etc.
- Manage the testing process as part of the development process: execute tests, file fault reports and direct them to developers, project or product managers, measure test result, and create test reports.
- Plan, organize, direct, and take responsibility for reviews and inspections of documents and code.
- Identify and use the tools appropriate for a particular situation and goal.
- Select test cases depending on probability of finding faults, associated risks, and required coverage.

Training

Training is a way to achieve adequate skills and knowledge. Three levels of tester training are developed and used.

- Foundation (this course)
- Advanced (three courses – Test Manager, Test Analyst and Technical Test Analyst)
- CTEL – Expert Level (four modules – Test Management, Improving the Testing Process, Test Automation Engineering and Security Testing)
- Extensions – Agile Tester and Model Based Testing (expected in 2015)

A standardized training program for education of testers is developed by experts in software testing from all over the world. This internationally operating organisation is the ISTQB. The ISTQB is the umbrella organization for the national testing boards, which have already been established in many countries across Europe and around the world.

The Syllabus prepared by ISTQB for the Foundation level course includes the following main chapters:

- Fundamentals of testing
- Testing throughout the software lifecycle
- Static techniques
- Test design techniques
- Test management
- Tool support for testing

The courses are delivered only by accredited training providers. The accreditation process is very strict, so only a few companies in Europe are accredited to provide courses based on the ISTQB syllabi.

The training closes with a one-hour final examination. The examination is delivered by a national or supra-national certification body. On successful completion of the test the qualification ISTQB Certified Tester Foundation Level is awarded.

Organizations

ISTQB — www.istqb.org
the **International Software Testing Qualification Board**.

Its mission is to organize and support the ISTQB Certified Tester certification scheme for testers. ISTQB provides the core syllabi and sets guidelines for accreditation and examination for the qualifications in the scheme. The Board consists of many national and regional testing qualification boards, that represent ISTQB in their countries or regions.

SEETB — www.seetb.org
the **South East European Testing Board** is member of ISTQB.

Its mission is to promote and represent the *ISTQB Certified Tester* in South East Europe and to organize the accreditation, examination and certification processes in the region.

The South East European Testing Board is responsible for Bulgaria, Romania, Serbia and Montenegro.

iSQI — www.isqi.org
the **International Software Quality Institute**. It is a subsidiary of the German non-profit organization for Software Quality ASQF. It provides services related to software quality - from personnel certification to international standardization. Their mission is to advance the field by coordinating efforts aiming at new, higher industry standards.

ISEB — http://www.bcs.org/server.php?show=nav.001010002
the **Information Systems Examinations Board** of the British Computer Society or BCS.

It has a purpose similar to that of ISTQB only that it is designed to serve the needs of the British Computer Society - "provide industry-recognized qualifications that measure competence, ability and performance in many areas of IS, with the aim of raising industry standards, promoting career development and providing competitive edge for employers."

You may find it mentioned at several places in the text. ISEB as a body is equivalent to ISTQB. It offers training and certification in the entire spectrum of IT skills and branches.

IEEE — www.ieee.org
the **Institute of Electrical and Electronics Engineers**. An American organization with 40% international members. Its mission is to promote the engineering process of creating, developing, integrating, sharing, and applying knowledge about electro and information technologies and sciences for the benefit of humanity and the profession.

1. Fundamentals of testing

1.1. Why is testing necessary

1.1.1. Software systems context

The complex nature of software that is built today and the speed with which the new software is required to be available, inevitably leads to mistakes / oversights occurring. Without undertaking any testing of the software, we would probably end up with it not meeting the user requirements or not even working.

We have all encountered problems with software that we use on a daily basis on our personal computers and this software has been subjected to extensive testing prior to release. Can you imagine the potential problems that we would all face if no testing of software had taken place prior to a release?

Testing software definitely reduces the likelihood of a failure and if it is undertaken in a professional manner, the software will meet all specified user requirements when it is delivered.

The Cost Of Errors

A single failure may incur little cost - or millions.

- Report layouts may be wrong - little true cost.
- Or a significant error may cost millions… Ariane V, Venus Probe, Mars Explorer and Polar Lander.
- In extreme cases a software or systems error may cost LIVES.

If a report displays dates as DD-MM-YY instead of DD/MM/YY it is an error, but the impact is minimal.

However, a single error has been known to cost millions as in the Space program examples.

Testing of safety critical software is of paramount importance. Failures in this software can cost lives - Air Traffic Control systems, aeroplane fly-by-wire control systems etc. Usually these systems are tested exhaustively.

Unfortunately, there are exceptions...significant errors that appeared in the London Ambulance Service system soon after implementation were identified as problems that ultimately resulted in the loss of human life.

The LAS have learnt from these harsh lessons and are now one of the better outfits in the UK.

The cost of defects increases proportionately (tenfold?) with the passing of each successive stage in the system development process before they are detected.

- To correct a problem at requirements stage may cost €1.
- To correct the problem post-implementation may cost €1000's.

The Cost of Errors will be looked at in more detail later in the course.

Over the past couple of years, testing has emerged for the following key reasons:

- e-Commerce.
- The Y2K problem.
- EMU.
- Increased user base.
- Increased complexity.
- Time to Market.

The item that developed the profile of testing the most was the Year 2000 problem. This prompted a significant number of companies to increase the profile of testing. The matter was further highlighted because legislation made it mandatory for all organisations to take positive measures to eliminate issues - Company Directors were to be liable.

E-commerce has further raised the profile of testing – too many web sites have been released with major, high profile faults in them.

The next major boost to testing in some of the European countries is the EMU. In addition to making software supporting dual currency, changes have to be made to supporting legislation.

Similar legally driven changes may come about with increased Data Protection laws. The others are evolutionary factors that have placed greater emphasis on testing - primarily lead by changes in technology.

Terms

Testing terms vary throughout the software testing industry and misunderstandings often occur as a result of problems related to the usage, understanding, or translation of the terms used. Various glossaries of testing terms and other sources of definitions exist. Some of them are:

- The British Computer Society (BCS) Special Interest Group in Software Testing (SIGiST) has provided definitions in the Standard Glossary of Testing Terms (British Standard BS7925-1).
- IEEE has a glossary on Software Engineering Terminology and a number of standards on testing in particular.
- ISO's series of standards define specific vocabularies.

The testing terminology used throughout this course has been taken from the ISTQB Standard **Glossary of Terms used in Software Testing** version 3.01 (March 2015), produced by the ISTQB 'Glossary Working Party'.

The ISTQB glossary is the latest work on testing terminology and incorporates all the three sources mentioned above.

The most basic terms will be discussed later in the course as they become relevant.

1.1.2. Causes of software defects

There are a variety of reasons as to why errors occur during software development.

- No one is perfect!
- We all make mistakes or omissions.

- The more pressure we are under the more likely we are to make mistakes.
- In IT development we have time and budgetary deadlines to meet.
- Poor Training.
- Poor Communication.
- Requirements not clearly defined.
- Requirements change & requirements not properly documented.
- Data specifications not complete.
- ASSUMPTIONS!
- A clear vision of the end game is not fully defined when development commences.
- The objectives are often redefined during the project's Life-cycle.

Making mistakes is a natural thing. Testing helps to find those mistakes. A Quality Assurance or Continuous Improvement program helps to ensure we learn from those mistakes and don't repeat the error in future. The pressure of a development project with time, budgetary constraints and deadlines will increase the likelihood of us humans making mistakes.

ERROR

"A human action that produces an incorrect result."

You will hear many different names for 'errors'… including Faults and Defects.

Errors can occur anywhere in the software development life cycle, they are not restricted to just developers' code.

DEFECT = FAULT = BUG = PROBLEM = ISSUE = ERROR CONDITION

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

Although all these words are often used to cover the same thing, there are certain nuances. For example, the term Defect can be taken to as being less emotive and the term 'bug' generally implies there is a fault in the software made by developers.

The following example illustrates the difference between defect and error:

Example:

The specification for a piece of software that does a calculation states that 2+2=5. The person who wrote in the specification that 2+2=5 would have made an ***error***. By doing so he would have created a ***defect***.

A Fault, if encountered, may cause a failure.

FAILURE

"Actual deviation of the component or system from its expected delivery, service or result."

Also called 'external fault'.

Inevitably, ALL Failures are caused by Faults. In some instances the actual failure may be more immediately obvious than in others. Not all failures are necessarily caused by software faults they can also be caused by hardware faults.

Faults, for example, may lie in the code that accumulates information for weekly or monthly reports, it may be possible for these to remain undetected until the report is produced and scrutinised.

ANOMALY

"Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience.

Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation."

This is the most general term.

DEFECT MASKING

An occurrence in which one defect prevents the detection of another.

Also called 'fault masking'

Some recent major computer system failures caused by software faults?

- According to newspaper stories in July and August of 2001, a major systems development contractor was fired and sued over problems with a large retirement plan management system. According to the reports, the client claimed that system deliveries were late, the software had excessive defects, and caused other systems to crash.
- In January of 2001 newspapers reported that a major European railroad was hit by the after effects of the Y2K bug. The company found that many of their newer trains would not run due to their inability to recognize the date '31/12/2000'; the trains were started by altering the control system's date settings.
- News reports in September of 2000 told of a software vendor settling a lawsuit with a large mortgage lender; the vendor had reportedly delivered an online mortgage processing system that did not meet specifications, was delivered late, and didn't work.
- In early 2000, major problems were reported with a new computer system in a large suburban U.S. public school district with 100,000+ students; problems included 10,000 erroneous report cards and students left stranded by failed class registration systems; the district's CIO was fired. The school district decided to reinstate its original 25-year old system for at least a year until the faults were worked out of the new system by the software vendors.

- In October of 1999 the $125 million NASA Mars Climate Orbiter spacecraft was believed to be lost in space due to a simple data conversion error. It was determined that spacecraft software used certain data in English units that should have been in metric units. Among other tasks, the Orbiter was to serve as a communications relay for the Mars Polar Lander mission, which failed for unknown reasons in December 1999. Several investigating panels were convened to determine the process failures that allowed the error to go undetected.
- Faults in software supporting a large commercial high-speed data network affected 70,000 business customers over a period of 8 days in August of 1999. Among those affected was the electronic trading system of the largest U.S. futures exchange, which was shut down for most of a week as a result of the outages.
- In April of 1999 a software bug caused the failure of a $1.2 billion military satellite launch, the costliest unmanned accident in the history of Cape Canaveral launches. The failure was the latest in a string of launch failures, triggering a complete military and industry review of U.S. space launch programs, including software integration and testing processes. Congressional oversight hearings were requested.
- A small town in Illinois received an unusually large monthly electric bill of $7 million in March of 1999. This was about 700 times larger than its normal bill. It turned out to be due to faults in new software that had been purchased by the local power company to deal with Y2K software issues.
- In early 1999 a major computer game company recalled all copies of a popular new product due to software problems. The company made a public apology for releasing a product before it was ready.
- The computer system of a major online U.S. stock trading service failed during trading hours several times over a period of days in February of 1999 according to nationwide news reports. The problem was reportedly due to faults in a software upgrade intended to speed online trade confirmations.
- In April of 1998 a major U.S. data communications network failed for 24 hours, crippling a large part of some U.S. credit card transaction authorization systems as well as other large U.S. bank, retail, and government data systems. The cause was eventually traced to a software bug.
- January 1998 news reports told of software problems at a major U.S. telecommunications company that resulted in no charges for long distance calls for a month for 400,000 customers. The problem went undetected until customers called up with questions about their bills.
- In November of 1997 the stock of a major health industry company dropped 60% due to reports of failures in computer billing systems, problems with a large database conversion, and inadequate software testing. It was reported that more than $100,000,000 in receivables had to be written off and that multi-million dollar fines were levied on the company by government agencies.
- In August of 1997 one of the leading consumer credit reporting companies reportedly shut down their new public web site after less than two days of operation due to software problems. The new site allowed web site visitors instant access, for a small fee, to their personal credit reports. However, a number of initial users ended up viewing each

others' reports instead of their own, resulting in irate customers and nationwide publicity. The problem was attributed to "...unexpectedly high demand from consumers and faulty software that routed the files to the wrong computers."

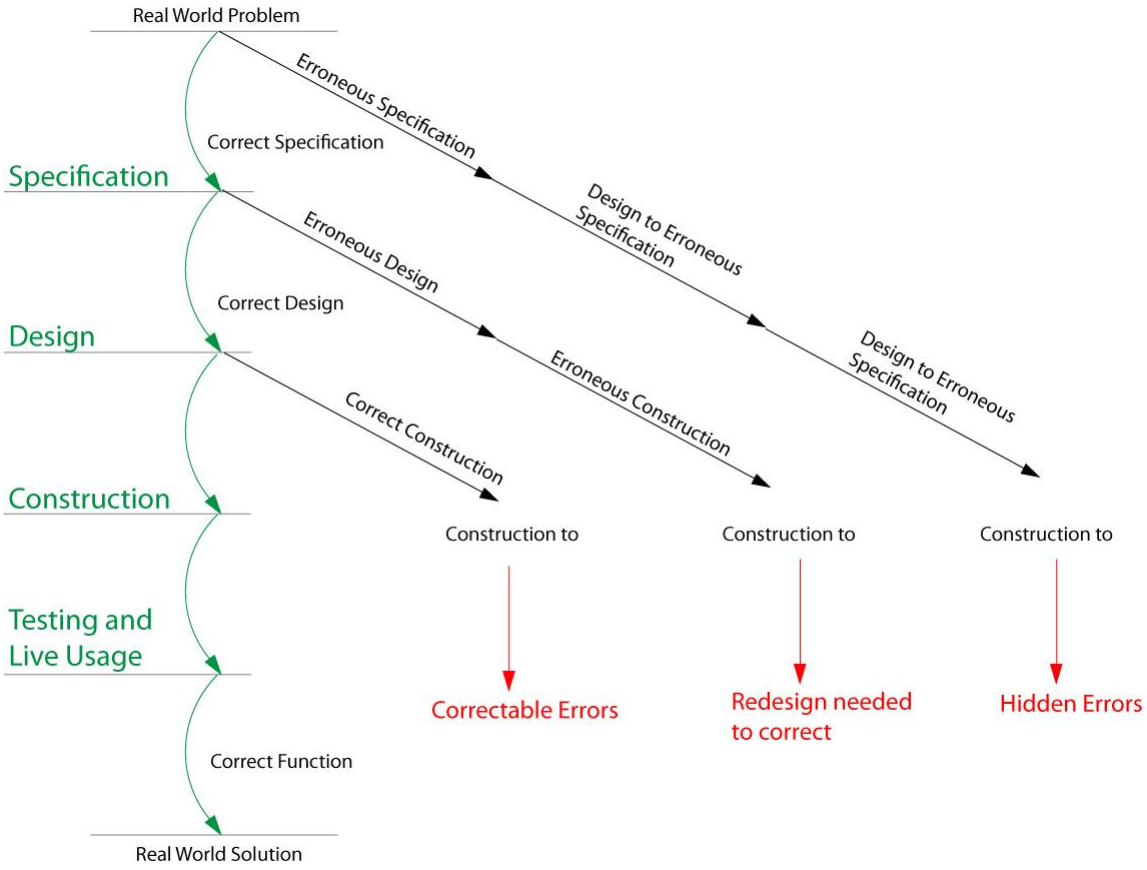
- In November of 1996, newspapers reported that software faults caused the 411 telephone information system of one of the U.S. RBOC's to fail for most of a day. Most of the 2000 operators had to search through phone books instead of using their 13,000,000-listing database. The faults were introduced by new software modifications and the problem software had been installed on both the production and backup systems. A spokesman for the software vendor reportedly stated that 'It had nothing to do with the integrity of the software. It was human error.'
- On June 4 1996 the first flight of the European Space Agency's new Ariane 5 rocket failed shortly after launching, resulting in an estimated uninsured loss of a half billion dollars. It was reportedly due to the lack of exception handling of a floating-point error in a conversion from a 64-bit integer to a 16-bit signed integer.
- Software faults caused the bank accounts of 823 customers of a major U.S. bank to be credited with $924,844,208.32 each in May of 1996, according to newspaper reports. The American Bankers Association claimed it was the largest such error in banking history. A bank spokesman said the programming errors were corrected and all funds were recovered.
- Software faults in a Soviet early-warning monitoring system nearly brought on nuclear war in 1983, according to news reports in early 1999. The software was supposed to filter out false missile detections caused by Soviet satellites picking up sunlight reflections off cloud-tops, but failed to do so. Disaster was averted when a Soviet commander, based on what he said was a '...funny feeling in my gut', decided the apparent missile attack was a false alarm. The filtering software code was rewritten.

1.1.3. Role of testing in software development, maintenance and operations

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

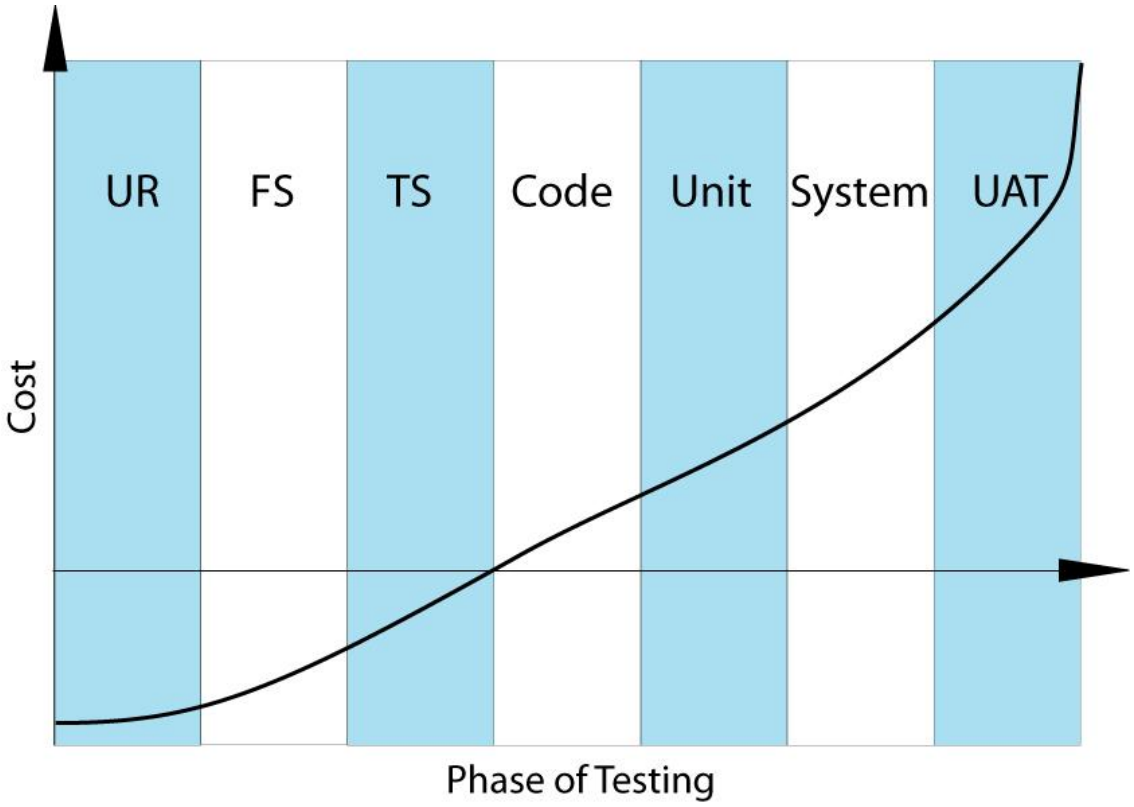
The Propagation of Errors

The following diagram illustrates the propagation of errors throughout the SDLC:



The Model of Cost Escalation

The cost of correcting a defect can be graphically demonstrated by the Model of Cost Escalation. The cost of correcting a defect during progressively later phases in the System Development Life Cycle (SDLC) rises alarmingly. It is generally accepted in the software testing industry that the cost of fixing a defect will increase by a factor of ten for each successive project phase.



Where:

- UR = User Requirements
- FS = Functional Specification
- TS = Technical Specification
- UAT = User Acceptance Testing

Legal Requirements

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

A common practice in many organisations is to outsource or subcontract all or a portion of the test activities to other companies that specialise in various aspects of software testing. Although, this might sound more cumbersome and more expensive than having the work undertaken by the organisation's own personnel, it can be an effective method to share the testing activities if done correctly.

However, if a company does choose to outsource all or a portion of their testing, then contractual requirements between the various parties become very important. The contract will need to detail the scope of the testing to be undertaken by each

of the parties, the tests to be performed (including the depth and breadth of the coverage), time frame, entry and exit criteria, pass / fail criteria, suspension / resumption criteria etc. In fact, the more detailed the criteria listed within the contract is, the greater confidence the organization can have that the testing that has been performed by the testing company, will meet their requirements for quality.

Many organisations also outsource their system development to outside contractors. These days, there is a greater requirement to get a system developed and available to end users in shorter and shorter timeframes and quite often the organisation's own development team does not have the required skills or resources to make this possible. In these cases the 'development team' responsible for writing the source code will usually do the low level unit / module testing and system testing leaving the client to undertake the user acceptance testing. This gives rise to three questions

- What testing has the development team performed?
- How detailed was their testing i.e. was it only positive testing to prove that the system worked or did they perform negative testing as well?
- Does the code that has been developed meet the business / user requirements that were specified at the beginning of the project?

The key to successfully outsourcing development and testing projects is to ensure that the contract between the parties contains sufficient details to prevent any misunderstandings /ambiguities arising.

Testing and Legal, Regulatory or Mandatory Requirements

Computer systems do fail, even after they have been thoroughly tested. However, if a system that has been subjected to little, poor or no testing fails, then there is no excuse if the Directors of the organization are held liable for its failure.

The risk of failure in a computer system is almost impossible to eliminate totally and as a result, it is important that testers can demonstrate that they have tested the system to the best of their ability and to the degree that the client specified within the contract. As previously mentioned, testing is a risk reduction / risk management process and the owner of a new computer system that has been subjected to testing needs to be aware that there are still potential risks associated with putting the system live (unless exhaustive testing has been achieved). The potential risk of the system failing should have been substantially reduced, if the level of testing specified within the contract has been adhered to. Although, the owner of the new computer system still needs to appreciate that there are risks associated with putting the system live. The decision to put the system live rests with the system owner and in effect they are stating that they are happy to accept the remaining potential risks associated with putting the system live.

Each testing project is different, as well as the levels / degrees of testing required for testing systems within different industry / business sectors are. For instance, it is important to exhaustively test a computer system that is being used to control the movement of trains on a particular stretch of track in order to eliminate all potential risks of failure. This might result in the testing of the system costing millions of pounds but when you consider the potential devastating impact that a failure of the system might cause, it is money that is well spent. However, the same degree of testing cannot be justified for the computer system that is used by the same train operator for the sale of tickets. A failure of the computer system for tickets sales will be inconvenient but it is not likely to result in the loss of life.

In some industries, there are Statutory / Legal requirements for the level of testing that must be undertaken and it is important that testers are aware of any Statutory / Legal requirements for a particular industry / project from the outset.

Both consumers and business community have in the past voiced concerns over the quality and reliability of computer systems that are currently being used. To address these concerns regulatory bodies have introduced testing standards like the ISO 9000 series. Moreover companies can now be graded to see where they fit into the Capability Maturity Model (CMM). Companies that are ISO compliant or a particular CMM level can be confident that their development processes will help quality to be in-built into the systems that they produce.

Test to ensure legal or statutory requirements are met.

- Contractual obligations to test software.
- Agreed best practices (non-negligent practice).

A number of industry sectors made demands on the software used to ensure certain actions happen at certain times. Banking applications in particular must ensure that the National Bank's Regulations aren't breached. Investment software must ensure Personal Investment Authority regulations are adhered to.

Gradually, legal precedents are being established to ensure software vendors do not market or continue to market sub-standard software or software with known defects.

1.1.4. Testing & Quality

The purpose of testing is to find faults in the SUT(System Under Test). The following benefits can be pointed out:

- Reducing the number of defects that are contained in the released version of the software.
- Improving the quality of the software - there will be fewer faults and a greater chance that the software will match its' specification.
- Increasing reliability - software with fewer faults is by nature more reliable.
- Providing a measure of the current quality of the software.

To demonstrate the level of quality for the SUT, testing must be carried out throughout the SDLC. Quality starts at the beginning of the project, i.e. when the user requirements are first defined and not when the code is first delivered. By starting the testing process at the earliest possible opportunity in the SDLC, testers can help to ensure that errors do not get propagated from one phase of the SDLC to the next, which will help to ensure that quality is built into the SUT.

QUALITY

> *"The structured set of characteristics of an entity that bear on its ability to satisfy stated or implied needs".*
>
> Quality is the sum of all characteristics of an entity that bear on its ability to satisfy stated or implied needs.
>
> Testing by itself does not build Quality into software, nor does it improve the quality of the SUT. Testing simply shows the quality of the software being tested. Testing is a means of determining the level of Quality of the Software under Test.

What is the difference between Testing and Quality, then? Quality and Testing used to, and to some extent still get confused. Quality relates to all entities that comprise a business solution, of which the testing of the software provides an integral part.

According to ISO 9126 (replaced by ISO 25010:2011) software quality, besides functionality, includes reliability, usability, efficiency, maintainability and portability/applicability.

RELIABILITY

"The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations."

Within commercial organisations, it is generally accepted that to carry out comprehensive testing is not practical. Safety critical and business critical applications will require considerably more testing. In the case of safety critical software, comprehensive testing is a must!

However, enough testing should be carried out in the time allowed to ensure that a profitable level of reliability can be achieved. That is, relatively low probability that under certain conditions the software will not cause failures of the system for a specified period of time.

It is not necessary that testing stops once an application is made live, testing can continue throughout an application's production life. This is an area that will be explored later in the course.

USABILITY

"The capability of the software to be understood, learned, used and attractive to the user when used under specified conditions."

EFFICIENCY

"The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions."

MAINTAINABILITY

"The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment."

PORTABILITY

"The ease with which the software product can be transferred from one hardware or software environment to another."

QUALITY ASSURANCE

"Part of quality management focused on providing confidence that quality requirements will be fulfilled."

Again, sometime ago Testing and Quality Assurance were often synonymous and in some instances still are.

QUALITY CONTROL

"Quality Control is an activity performed to check that a product is fit for purpose."

There is more ground to equate testing with quality control than with quality assurance, yet quality control is not the same as testing.

Software quality control has two aspects, and testing is related to the second one:

- **constructive** quality control: focused on avoiding defects.
- **analytical** quality control: focused on finding defects.

Quality Measurements

We can measure the quality of software by measuring the relevant critical factors:

- Correctness - how accurately the software performs the functions defined in the specification.
- Reliability - the time or transactions processed between failures in the software (mean time to failure).
- Usability - the ease of use of the software by the intended users.
- Maintainability - how easy it is for developers to maintain the application and how quickly maintenance changes can be made.
- Reusability - how easy it is to re-use elements of the solution in other identical situations.
- Testability - how easy it is to test the application, clear, unambiguous requirements.
- Legal requirements.

The IS0 9000 Series

The ISO is the International Organisation for Standardisation and is based in Switzerland.

ISO 9000

The standard ISO 9000 embodies the basic requirements for a quality management system based on a cross-section of expert advice from all sectors of industry.

This is a minimum of:

- Best Practice
- Description of control activity to ensure compliance
- Formal documentation to demonstrate compliance

There is the notion built into ISO 9000 that the organisation certified to the standard will be able to demonstrate that if defects are discovered then corrective action is to be taken to reduce the likelihood of recurrence of those or similar defects.

The standard requires that there is testing specified for all:

- Input
- Processes
- Output

The coverage must be thorough and must contain a quality improvement mechanism. The key sections of the standard are:

- Statement of Best Practice
- Statement of how Best Practice products are reviewed
- Statement of how the review process maintains quality
- Statement of how quality achieved will be demonstrated to customers
- Statement of a quality improvement activity / process.

ISO 9000 has become mandatory for a considerable number of businesses. The standard requires compliant organisations to provide evidence of correct working of

all elements of the standard. Gaining **ISO 9000 compliance is not a guarantee of quality, it merely provides for reproducibility.**

The particular parts of the ISO standard that relate to testing are:

- ISO 9001: Quality Systems - Model for quality assurance in design, development, production, installation and servicing.
- ISO 9000 - 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.

ISO 9126

This series defines the quality characteristics of software at the life cycle stages.

Part 1 - Quality Model - defines the criteria for good software:

- Functionality (does it do what you want?)
- Reliability (all the time, every time?)
- Usability (without a PhD in computer science and years of training?)
- Efficiency (with a reasonable amount of computer resources and time?)
- Maintainability (can it be upgraded?)
- Portability (can it work on different platforms?

Part 2 - External Use - defines external metrics that measure the behaviour of the computer-based system that includes the software; in other words, measure the "goodness" of the behaviour of the software system, after it is done.

Part 3 - Internal Use - defines the internal metrics for measuring the software itself.

Both parts 2 and 3 break down the quality characteristics defined in part 1 into measurable tasks. So 9126 Part 2 External Functionality is divided into:

- Suitability
- Accuracy
- Interoperability
- Security
- Functionality compliance

Part 4 - Quality in Use metrics - measures the effects of using the software in a specific context of use; that is, if the software meets the needs of the specified users and to what extent it met the specified goals for effectiveness, productivity, safety, and satisfaction.

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 reoccurring and, as a consequence, improve the quality of future systems.

Process Improvement

Some of the most widely used models are detailed below:

- ISO/IEC 15504 Software Process Improvement and Capability dEtermination (SPICE)
- Software Engineering Institute (SEI) Capability Maturity Model (CMM®)
- SEI Capability Maturity Model Integration (CMMI)
- Illinois Institute of Technology (IIT) Testing Maturity Model (TMM®)
- Sogeti B.V. Test Process Improvement (TPI®)

All of the models listed have two objectives in common:

1. Assessment of an existing process
2. The improvement of an existing process

Some models, i.e. SPICE assess the "Capabilities" of a process, which in turn identify areas for improvement. Others assess the "Maturity" of a process, give it a "Level", and define what has to be achieved to progress from one level to another.

ISO / IEC 15504 SPICE

The primary objective of SPICE is to assess a software process, and determine what its "Capability" is. The outcome is to assign one of the 6 "Capability Levels". Process Improvement can be achieved after areas with low capability levels have been identified. This approach also allows a comparison of various business areas to be made between different companies - for example the implementation for automotive industry called Automotive SPICE or the next variation – Banking SPICE.

Capability Maturity Model (CMM®)

CMM is also known as the Capability Maturity Model for Software (SW-CMM®). The CMM model describes software process maturity, assesses a process to determine how mature it is, and enables improvement of its maturity.

CMM has five maturity levels, which range from "Initial" where none or few processes are defined, to "Optimising" where defined processes are improved by quantitative feedback.

The goals that have to be achieved in order to move up to a higher level of maturity are defined in the model.

Capability Maturity Model Integration (CMMI^{SM})

CMMI is a set of tools and products, which enable the Integration of CMM. The model has been updated to combine three models, which will now be phased out:

1. SW-CMM
2. EIA/IS 731 Systems Engineering Capability Model
3. Integrated Product Development CMM

CMMI also includes two Representations:

1. *Staged* which concentrates on "Maturity Levels", therefore it is similar to CMM.
2. *Continuous* which concentrates on "Process Areas", therefore it is similar to SPICE.

Testing Maturity Model (TMM®)

All of the previous models apply to software development, and do not adequately address Software Testing. TMM has been developed to focus on testing. It is very similar to CMM, and compliments it, rather than replacing it in a software development organisation.

Test Process Improvement (TPI®)

TPI is another process improvement model that concentrates on testing. This model identifies "Key Areas" within a process, and assesses their maturity levels.

Once areas for improvement have been identified, the emphasis moves more towards suggested improvements and activities to improve the maturity levels.

1.1.5. How much testing is enough?

How do we know when to stop?

To test everything is rarely possible. The time and resource commitment required makes it impractical.

Only in very few cases it is possible to test every aspect of a piece of software. A comprehensive or exhaustive test would need to cover every possible combination and permutation of conditions and data.

The two main constraints on the level of testing performed are time and resources. The time and resource needed simply to plan and prepare such a range of tests is prohibitive, even before we consider the other testing activities to be carried out (Execution and Verification).

> **Example:**
>
> A system has 20 independent inputs, each of which has four possible values.
>
> To exhaustively test it we would need 4^{20} = 1,099,511,627,776 tests
>
> If each test takes 1 minute, to run all tests it would take up one person working continuously just over 2 million years

Some systems will require greater levels of testing - those that must match legal requirements and those that risk life if failures occur.

We need a way to reduce the amount of testing that is essential before software is made live.

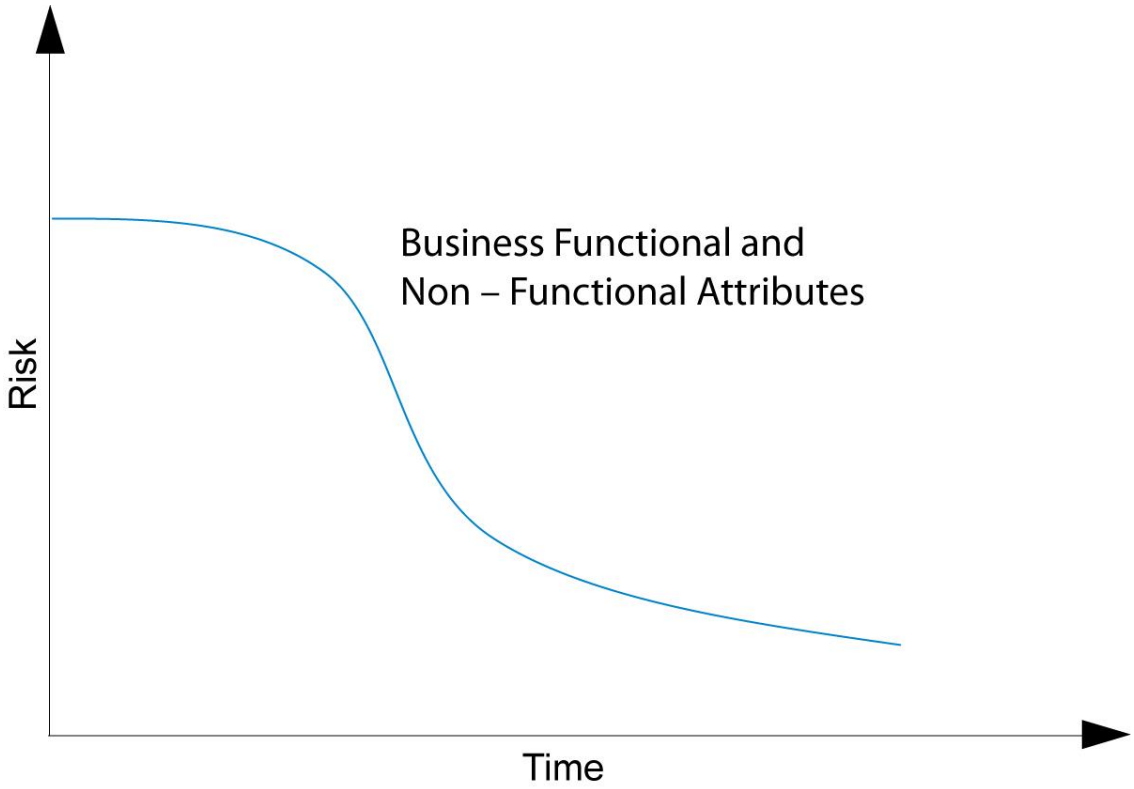
One of the fundamental methods that can be utilized to reduce the number of tests that need to be run is prioritising the risk factor associated with particular functions / business requirements of the SUT. However, before any tests can be prioritised, it is of paramount importance to understand the scope and the business requirements of the SUT. Unless the tester has a clear understanding of these basic facts, it is impossible to accurately prioritise which areas to concentrate on when commencing the testing of the SUT.

We will cover prioritisation of tests and the factors to be considered later in this course.

Risk Reduction

Each software development project has its own unique risk factors that testers will need to consider and plan for when they undertake their task of testing the SUT. There are many different types of risk that may be connected to a software development project.

Risk Reduction Profile



The risk can be defined as a combination of two factors, the probability of a problem to occur and the impact that this problem will inflict.

Testing is an exercise in risk management and reduction.

As testing is normally unable to cover every aspect of the SUT, it is necessary for the testers to concentrate their limited time and resources on certain areas of the software.

The decision as to which areas to test is normally based upon the risk involved. The higher the risk to the business the greater priority the testing of a given area will be.

Carrying out a Risk Analysis of the application will enable us to identify those areas of the application that are critical to the continued functioning of the business. Testers can then prioritise tests to focus on the main areas of risk and apportion their time relatively to the extent of the risk involved.

"Risk" can be represented as Critical Success Factors ("CSFs") - what factors are critical to the success of the business. By identifying their elements we should be able to prioritise tests for new applications.

If the business users can say "If Function X does not work it will not be worth launching the product", you will need to ensure that your tests exercise Function X comprehensively before launch.

Exhaustive testing

Exhaustive testing is the execution of a program with all possible combinations of inputs or values for the program variables. However, exhaustive testing is impossible. Therefore we are unable to fully test a system. So how do we know how much testing is enough?

There are many factors to consider - the most important of which are risk, resources and time.

The major issue is risk. Those areas of the system deemed critical to the business must be thoroughly and repeatedly tested. The amount of time and resources available to test will affect the amount of testing that can be done.

If an area is untested or has significant faults then a decision as to whether to deploy the software must be made by the business. The business must decide whether the risk is worth taking or whether the system deployment should be delayed until the risk is reduced.

Weighed against this may be the risk of missing a business opportunity and / or public embarrassment because a previously published date cannot be kept to.

How do You Identify What Needs to be Tested?

The answer to this question is that the user requirement, business requirement, system specification and other available documents should provide sufficient details for the tester to ascertain the scope of the project and help to identify a suitable approach. This will enable the tester to construct a suitable Test Plan which will detail those functions / areas to be tested and those functions / areas that will not be tested. The Test Plan will be checked / agreed / signed off by the client prior to implementation. This should ensure that the tester has thoroughly understood the scope of the project / testing required and ensure that sufficient coverage for testing the processing rules / testing criteria has been identified. If the tester has adequately identified all the relevant risks involved within the system and has planned sufficient tests to cover these risks and once the test criteria have been met, the system can be deemed 'fit for purpose'.

How do You Know When to Stop Testing?

When the exit criteria for a phase of testing have been met, the risk associated with putting that part of the system live should have reached an acceptable level. Testing as mentioned earlier is a risk management process and by undertaking it to achieve acceptable exit criteria prior to putting a system live, the risk of live failure should be greatly reduced.

Testing also demonstrates the quality of the SUT and by adhering to the exit criteria agreed within the Test Plan, the right level of quality will be built into the software.

The theoretical / ideal answer to this question is that it depends on the risk to the business.

Testing When the Product is Live

Why stop testing when the product is live?

Continuing testing beyond the launch date should be considered. It is better if you find the errors than the real users!!

This is particularly important for Web Based applications where users will - if they find an error in your Web Site that prevents them getting what they want - quickly go to another site and carry out the business with another supplier. "Voting with the mouse!"

1.2. What is testing?

What is Testing?

There are various definitions of testing; some of the more popular ones follow.

Here is one definition of testing that is often quoted:

> *"…the process of executing a program with the intent to certify its Quality"*
>
> Mills

It does not explicitly highlight the fact that testing discovers errors.

This is another definition often used to describe testing:

> *"…the process of executing a program with the intent of finding errors."*
>
> Myers.

This one does not indicate that testing checks adherence to requirements. By implication, we could use it as our parameter for testing and deliver fault free applications that serve no business purpose whatsoever!

To get a good definition of testing we need to combine elements of the Miles and Myers definitions, to present what we believe to be testing:

> *"…the process of exercising software to detect errors & to verify that it satisfies specified functional & non-functional requirements."*

BS7925-1:1998 gives the following definition of testing:

> *"…the process of exercising software to verify that it satisfies specified requirements and to detect errors."*

The ISTQB definition of testing is:

> *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.*

TEST

> *A set of one or more test cases*

TEST CASE

> *A set of input values, execution preconditions, 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.*

DEBUGGING

> *"The process of finding, analyzing and removing the causes of failures in software."*
>
> The boundaries between testing and debugging are not always clear. Testing is intended to directly and systematically discover failures, which point to defects. Debugging is intended to locate the defect in order to remove it.

A common perception of testing is that it only consists of running tests, i.e. executing the software. This is one part of testing, but not all of the testing activities.

Within any testing activity we need to ensure that the software being tested is adequately exercised to uncover any errors that exist within it, but also confirm that it does what it has been specified to do not only in terms of the function or service it provides but also within the constraints - or non-functional requirements – that have been placed on it.

Testing is a part of the SDLC before and after test execution. It includes activities such as test planning and control, choosing test pre-conditions and post-conditions, designing test cases and checking results, evaluating test completion criteria, reporting on the testing process and SUT, and finalizing or closing of the test project (i.e. after a test phase or the entire test project has been completed). A part of testing is also the review of documents (including source code) and the static analysis.

Dynamic testing and static testing can be used to achieve similar objectives, and will provide information in order to improve the system to be tested, and the development and testing processes. Static testing is the term given to reviews and inspections of project deliverables (e.g. requirements specifications, design specifications, code, test plans) and provides a qualitative and quantitative assessment of the product tested. Dynamic testing on the other hand looks at how the software performs when the code is executed. The use of static or dynamic testing in isolation is unlikely to give us the most robust solution; each should be used appropriately throughout the development life-cycle.

Four test levels can be distinguished: component, integration, system, and acceptance testing. In each of the test levels different viewpoints can be considered as important. For example, in component, integration and system testing the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. In acceptance testing, on the other hand, the main objective will most probably be to confirm that the system works as expected, to gain enough confidence that it has met the requirements. In other cases, however, the goal of testing may be to assess to what extent the quality characteristics of the software are fulfilled (with no fixing activities involved), to give information to stakeholders of the risk of releasing the system at a given time.

Different organizations and different individuals have varied views of the purpose of software testing. Boris Beizer describes five levels of testing maturity.

Level 0 - "There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose."

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 confirmation 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.

Level 1 - "The purpose of testing is to show that software works." This approach, which starts with the premise that the software is (basically) correct, may blind us to discovering defects. The people performing the testing may subconsciously select test cases that should not fail. They will be inclined not to create tests needed to find deeply hidden defects.

Level 2 - "The purpose of testing is to show that the software doesn't work." This is a very different mindset. It assumes the software doesn't work and challenges the tester to find its defects. With this approach, we will consciously select test cases that evaluate the system in its nooks and crannies, at its boundaries, and near its edges, using diabolically constructed test cases.

Level 3 - "The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value." While we can prove a system incorrect with only one test case, it is impossible to ever prove it correct. To do so would require us to test every possible valid combination of input data and every possible invalid combination of input data. Our goals are to understand the quality of the software in terms of its defects, to furnish the programmers with information about the software's deficiencies, and to provide management with an evaluation of the negative impact on our organization if we shipped this system to customers in its present state.

Level 4 - "Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort." At this maturity level we focus on making software more testable from its inception. This includes reviews and inspections of its requirements, design, and code. In addition, it means writing code that incorporates facilities the tester can easily use to interrogate it while it is executing. Further, it means writing code that is self-diagnosing, that reports errors rather than requiring testers to discover them.

1.3. General testing principles

A number of testing principles have been suggested over the past 40 years and offer general guidelines common for all testing.

Principle 1 – Testing shows presence of defects

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

Principle 2 – Exhaustive testing is impossible

Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risk and priorities to focus testing efforts.

Principle 3 – Early testing

Testing activities should start as early as possible in the software or system development life cycle, and should be focused on defined objectives.

Principle 4 – Defect clustering

A small number of modules contain most of the defects discovered during pre-release testing, or show the most operational failures. This is yet another appearance of the 80/20 rule – 80% of defects come from 20% of the features.

Principle 5 – Pesticide paradox

If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To keep finding bugs, testers need to:

- review and revise existing test cases,
- add new and different tests to exercise different parts of the software.

An extension to the paradox is that developers gradually learn what kinds of bugs testers catch and start to make less of them. To avoid this, testers need to:

- learn new tools and methods and invent new ways to test,
- rotate so that the same tester does not work for a long time with the same developer.

Principle 6 – Testing is context dependent

Testing is done differently in different contexts. The context includes type of the product, its goals, associated risk, available tools, resources, and time, expertise of the team etc. For example, safety-critical software is tested differently from an e-commerce site; when you have 10 days and 5 testers you test differently compared to the way you test when you have 30 days and 2 testers.

Principle 7 – Absence-of-errors fallacy

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

1.4. Fundamental Test Process

The test process is an approach to testing which improves both development and testing

What is the objective of a test?

A "successful" test is one that does detect a fault i.e. it has performed its job.

- This is counter-intuitive, as faults delay progress.
- Thus a successful test is one that may cause some delay.

However,

- The successful test reveals a fault which, if found later, may be many times more costly to correct - so, in the long run, it is a good thing.

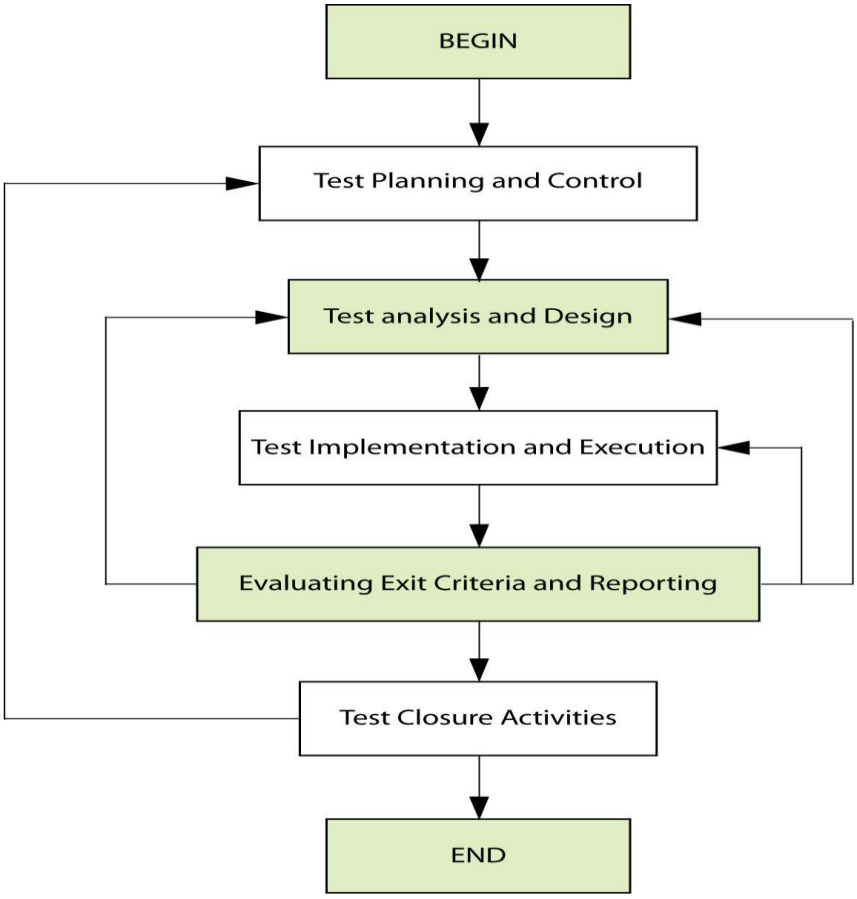
In very simple terms, Testing is a 'RISK MANAGEMENT' activity. It is carried out to reduce the risk involved in a software implementation.

We all take risks in life, some more than others. To many it is an acceptable risk to exceed the speed limit on the roads. Similarly organisations consider it an acceptable risk to implement new or changed software that has not been 100% tested. Naturally in either case the greater the risk you take the more likely your chances of being caught and having to pay the price.

There are five defined steps in the Test Process:

- Test planning, control and monitoring
- Test analysis and design.

- Test implementation and execution.
- Evaluating exit criteria and reporting
- Test closure activities.



We shall examine each of these steps in turn in the paragraphs that follow.

1.4.1. Test planning and control

The test plan should specify how the test strategy and project test plan apply to the software under test. This should include identification of all exceptions to the test strategy and of all software with which the software under test will interact during test execution, such as drivers and stubs.

The core activities that make up the test planning process are as follows:

- Define the test strategy
- Determine the scope and risks
- Determine the test approach (techniques, test items, coverage, identify the teams involved in testing)
- Schedule test analysis and design task
- Schedule test implementation, execution and evaluation
- Define the test system (hardware and software)
- Estimate the test effort (resources and schedule)
- Assess risks to the schedule and prepare mitigation plans
- Prepare and review the test plan documents

The test strategy is a high level document that includes the overall scope of testing, the types of testing methodologies that are to be used to find defects, the procedures for reporting and fixing defects and the entry / exit criteria that will govern the testing activities.

The test system refers not only to the hardware that is to be used in testing, but also the test architecture, the test tools, and the test configurations.

The major tasks concerning test control and monitoring are:

- Measuring and analyzing results;
- Monitoring and documenting progress, test coverage and exit criteria;
- Assign extra resources
- Re-allocate resources
- Adjust the test schedule
- Arrange for extra test environments
- Refine the completion criteria

1.4.2. Test analysis and design

Test analysis and design is the activity where general testing objectives are transformed into clear and concrete test conditions and test designs.

The application should be analysed in detail from both a user's perspective, and from a technical standpoint.

The art of testing is in analysing the SUT to produce and specify good tests. This will enable you to identify the conditions that need to be exercised during test execution.

Test analysis and design concentrates on the following tasks:

- Reviewing the requirements, architecture, design, interfaces.
- Identifying test conditions or test requirements and required test data based on analysis of test items, the specification, behaviour and structure.
- Designing the tests.
- Evaluating testability of the requirements and system.
- Designing the test environment set-up and identifying any required infrastructure and tools.

1.4.3. Test implementation and execution

The main objective in test implementation is to create detailed test scripts to exercise the criteria, in an efficient manner, which will achieve maximum coverage with the minimum number of test cases / use the minimum amount of test data. The reason that we do this is because test case design can be the most time consuming and labour intensive activity that is undertaken within the testing process. By creating scripts in this way, time is saved both during the design phase and also in the execution phase, as testers have fewer scripts to execute.

Additionally as part of defining the test cases testers would cross-reference wherever possible.

Test implementation and execution has the following major tasks:

- Defining and prioritizing test cases, creating test data, writing test procedures and, optionally, preparing test harnesses and writing automated test scripts.
- Creating test suites from the test cases for efficient test execution.
- Verifying that the test environment has been set up correctly.
- Executing test cases either manually or by using test execution tools, according to the planned sequence.
- Logging the outcome of test execution and recording the identities and versions of the software under test, test tools and testware.
- Comparing actual results with expected results.
- Reporting discrepancies as incidents and analyzing 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).
- Repeating test activities as a result of action taken for each discrepancy. For example, re-execution of a test that previously failed in order to confirm a fix (confirmation testing or re-testing), execution of a corrected test and/or 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).

1.4.3.1. Prioritizing the Tests

You need to ensure that when the application is made live you, the testers, have made every effort to ensure that critical elements have been tested as much as possible in the time available.

There are aspects of the application that are crucial to the success of the project or more importantly the survival of the company.

Exhaustive Testing

Ideally we want to exhaustively test the system - test every function in every possible way using every possible data variant. For safety critical systems - aeroplane control systems, nuclear power station systems etc - this is a MUST.

For commercial systems exhaustive testing is still the ultimate goal. However, the time and resources available for testing will make this impossible. Therefore, systems will be released without them having been completely tested.

What to do Instead?

Any system that is released without having been exhaustively tested runs the risk of containing faults. These faults could be minor (spelling errors) or could be such that they will crash the system and even prevent the business from carrying out its core tasks.

Testing is risk management - therefore we need to use testing to ensure that when the system goes live there is the least risk of it containing any catastrophic faults.

In order to minimise the risk we need to prioritise those parts of the system whose failure presents most risk to the continued successful functioning of the business.

Ultimately, we must prioritise tests so that whenever you stop testing you have done the best testing within the available time period.

What to prioritize?

As part of the test plan we need to identify those areas that present the maximum risk to the successful use of the software and ensure that we have sufficient tests to thoroughly test these areas.

There are a range of factors that must be considered when determining the priority of tests. For each system these factors will need to be given a ranking to further help with the prioritisation.

Severity

Tests need to be prioritised with consideration to the risk the organisation is exposed to should a particular function fails. The element, the malfunction of which will leave the organisation 'exposed' to a failure either through loss of customers or litigation, must be tested first.

Probability

Should there be a strong probability of a fault occurring within a function then tests should be created for that part of the system.

Visibility of failure

An error may not be severe in terms of its impact on the process, but if it is highly visible to the customer and/or occurs frequently, customers will regard you or your service in a poor light.

Priority of requirements

What functionality is crucial to the success and survival of the organisation?

Customer requirements

Having addressed basic business critical elements you look at the product through the eyes of a customer or user. What does the customer like or want in the application? Ensure these elements are tested.

Frequency of change

If code is subject to frequent change - try, if not covered already, to include these elements in your tests. The more often a specification is changed the more often the corresponding code will change and therefore the greater the risk of mistakes being made.

Vulnerability to error

It is possible that certain parts of a system may have an inherent vulnerability to error - these areas need to be tested.

Technical criticality

There may be aspects where the technical infrastructure is critical, especially these days when different platforms are used to service an application.

Complexity

Code that is complex to create or maintain will often be equally complex to test, if any of these aspects are not yet covered plan these. This also applies to complex hardware and networking configurations.

Time and resources

The two major issues that affect the definition of priority are time and resources.

As time grows less, so the definition of priority will narrow to include only those things that are critical to the survival of the business.

The number of resources available to testing will influence the scope of priorities - more testers should mean that more tests can be created & run and therefore more parts of the system can be given priority.

1.4.3.2. Building Test Cases

A test case comprises:

- Test Condition defined as a description in English of the functionality and the purpose of the test.
- Standing Data i.e. Data that needs to be in place prior to the test being applied. It can comprise several forms as depicted below:
  - Post Code reference file
  - Accounting period dates
  - Customer/Supplier Information
  - Product Information
- Transactional Data i.e. Data that needs to be processed during the course of a test to produce a result. This may take the form of:

  An order placed by Joe Bloggs of SL6 2HG, for 22 red-widgets, as supplied by BigWidge Co. to be supplied before end of the Tax Year.

Before running the test, we need to ensure that the necessary elements are present in the Standing Data (or Test-Bed).

- Actions or steps. Actions to be taken to perform the test need to be defined. For example:
  1. Enter Customer Name & Post Code.
  2. Select item & quantity to order.
  3. Select delivery dates.

1.4.3.3. Defining Expected Results

The purpose of testing is to find errors. When creating test cases you need to define what you expect to happen, before you execute the test.

If the test case simply states what to do and no expectations are set, how do you know whether or not the test passed or failed?

Expected results need to be documented to allow them to be compared with ACTUAL results. It is less time consuming to define the expected results when preparing the tests than to wait until later in the process.

All aspects of testing should be documented, reviewed and secured for future use and re-use.

The "**oracle assumption**" is that a tester can routinely identify the correct outcome of a test.

Without clearly defined expected results how are we able to determine if the test passed or failed?

By identifying the expected results (and therefore the expected behaviour of the system) we are able to determine whether or not the test passed. It is vital when designing tests - specifying what actions to be taken using what data - that the expected result of each action is clearly defined.

If the test is run with no clear definition of the outcome, the output of those actions may appear to be reasonable and accepted as such. This may be wrong and a defect may go undetected.

If we only have a rough idea of the expected result we are risking incorrect interpretation of the system performance and therefore incorrectly consider the test as passed or failed.

Example:

When planning a test you might have a script that says "System Date is 12/12/1999. Receive incoming package of SANDALS from supplier and <locate> in warehouse". Your expected results direct the stock to a warehouse location well away from the loading bay (demand for sandals in December is generally low) this may well be Pier 37 Bin 3.

If you don't define the expected results clearly, it is possible that when the test is run the result may look right but actually be wrong and the test incorrectly passed by the tester. It is therefore possible for the goods to be located in Pier 3 Bin 37 - this looks correct (it is in the correct format) but is actually an incorrect result.

The location looks right and, because of the vague expected results, the tester PASSES the test.

In essence this happened to a well-known shoe-retailing organisation. The error was not spotted until a shop in Aberdeen was unloading a delivery before Christmas to find boxes of lightweight sandals.

Expected results should be identified from the following sources:

- Knowledge or experience of the application.
- The system specification.

They should not be taken from examination of the code. Doing so would risk a test passing when in fact it should have failed because the code is incorrect.

1.4.3.4. Test Execution Checklist

Once the test cases have been created they are ready for execution. However, before they are executed a number of items need to be in place.

An MOT for a car is not a simple matter of checking out some stuff. The mechanics are armed with a prescribed list of items to check, in certain instances there is a logical sequence to these tests. Certain checks require elements of hardware (e.g. Emission Sensors etc,). The trained mechanics are in place and ready to carry out the tests. Each test that they carry out also has a prescribed expected outcome, acceptance criterion. If any of these are not met your car will fail the MOT.

Test Execution schedule / log defines the sequence in which tests are to be applied. The sequence depends on priority and the natural order in which the test should be carried out. The log is used to record:

- Date and time of test execution.
- Result of the test (pass or fail).
- Who ran the test?

- Schedule for re-tests.
- Defect reference for failures.
- Comments & Observations.

The test environment needs to be set up and ready for testing to satisfy the standing data requirements identified during test case analysis and preparation.

All support personnel & testers need to be available as identified during the definition of the test plan, and test case analysis and preparation.

1.4.3.5. Cross Referencing and Classification

When the Actual Result differs from the Expected Result, the tester must not assume that the defect lies within the application software. Typically for a first test of an application the percentage of errors caused by defective test specifications can be as high as 15-20%.

Therefore the tester must check to ensure that the test is right. To do this (s)he needs to refer back from the test data to the test condition and the systems documentation from which it was derived. This cannot be done unless there is a reference from the test data to the test condition to the source documentation.

Over time, applications requirements change, often during development. When such change occurs, the test analyst needs to identify, from the amended documentation what test conditions need to be revisited to assure that they are still relevant and that the detailed data conditions are still valid.

This also cannot easily be achieved without a reference with the conditions to the source documentation and also to the standing and transactional data.

Classification of tests helps greatly, tests should be classified as:

- Positive / Negative
- Functional / Non-Functional
- Etc.

1.4.3.6. Recording the outcome of test execution

Once a test has been executed its results should be recorded - did it pass or fail?

The test results for each test case should unambiguously record:

- The identities and versions of the software under test and the test specification.
- The actual outcome.

It should be possible to establish that all of the specified testing activities have been carried out by reference to the rest records.

The actual outcome should be compared against the expected outcome. Any discrepancy found should be logged and analysed in order to establish where its cause lies and the earliest test activity that should be repeated e.g. in order to remove the fault in the test specification or to verify the removal of the fault in the software.

The test coverage levels achieved for those measures specified as test completion criteria should be recorded.

Test Verification

Testing is an iterative process, tests that fail need to be recorded, corrected and re-tested repeatedly if necessary, until all the major defects have been resolved, in line with the predefined Exit Criteria.

1.4.4. 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 and is used to determine when testing (at any stage) is complete.

Evaluating exit criteria has the following major tasks:

- Checking test logs against the exit criteria specified in test planning.
- Assessing if more tests are needed or if the exit criteria specified should be changed.
- Writing a test summary report for stakeholders.

The exit criteria may be defined in terms of cost, time, faults found or coverage criteria.

Coverage criteria are defined in terms of items that are exercised by test suites, such as branches, user requirements, and most frequently used transactions etc.

The test records should be checked against the previously specified test completion criteria. If these criteria are not met, the earliest test activity that must be repeated in order to meet the criteria should be identified and the test process should be restarted from that point.

It may be necessary to repeat the Test implementation activity to design further test cases to meet a test coverage target.

Completion or exit criteria are used to determine when testing (at any stage) is complete. These criteria may be defined in terms of cost, time, faults found or coverage criteria. Coverage criteria are defined in terms of items that are exercised by test suites, such as branches, user requirements, and most frequently used transactions etc.

Meaning of Completion or Exit Criteria

How can we ascertain that a system is ready to go from one stage of testing to another or even ready to go 'live'? This is where defining / adhering to stringent completion or exit criteria is so important. The completion or exit criteria / conditions of a particular testing phase are identified and defined during the planning process for testing the various functional elements of the SUT. Until the completion / exit criteria have been successfully met for a particular testing phase, testing should not progress to the next phase.

Specific test conditions will relate to each separate phase of testing. For instance, the test conditions for User Acceptance Testing will have been derived from the User Requirement documentation.

The entry criteria for a particular stage of testing will include the exit criteria from the previous stage of testing e.g. the exit criteria from Unit Testing will be included within the entry criteria for System Testing. This effectively is a true quality control point.

Coverage Criteria

The test conditions that are identified in the planning stage must be cross-referenced back to the source documentation. The test conditions are also

cross-referenced to the scripts that have been written to verify them. This means that at any point during testing the coverage can be monitored in terms of what has been tested, what has yet to be tested and what has passed and failed.

The Deliverables

Plans are made on deliverables so it is necessary to construct a comprehensive list of deliverables first. The description of each deliverable identified needs to be reviewed against the list of the features that are fundamental to the business (i.e. business critical), items that are important but not critical and the 'nice to have' items. Each deliverable identified requires that strict acceptance criteria are set which are both measurable and comprehensively described.

As well as identifying / drawing up a deliverables list, the test team will need to construct a deliverables dependency matrix which will allow the cross reference of one deliverable with others. The deliverables dependency matrix will identify the areas where acceptance work can be undertaken and in which order the work should be done.

It is important that the person(s) responsible for this work has an understanding of how long these tasks will take and this can be obtained from the metrics database. However, if a metrics database is not available, they will have to go through the process of estimating and the process should be based on the company's standards.

The Overall Plan

Once the deliverables list, deliverables dependency matrix and the estimating have been completed, the test plans can be created. It is important to remember that plans only represent intentions and it is down to the team leader / Project Manager to ensure that what is detailed in the plans actually happens.

The Test Plan

Activities	Time
Business Study	XXX
Requirements Analysis	XXX
User Level Design	XXX
Technical Design	XXX
Program Specification	XX
Unit Test Planning	X
Creation of Code	XXX
Unit Testing	XX
Integration Test Planning	XX
Integration Testing	XX
System Test Planning	XX
System Testing	XX
Acceptance Test Planning	XX
Acceptance Testing	XXX
Implementation	XXX

Plans will be need to be grouped and they will be dependent on many fixed national (i.e. Public Holidays), company and project specific dates. The project planner will need to take all of these dates / dependencies into account but on no account, should the planning process be constrained by a delivery date. If the planner schedules the work around an end date, then it is not proper planning but is actually scheduling, which is the activity of fitting the planned work to be completed by a specific date.

The Test Log

The test log is the detailed record of the tests that have been carried out and it is the primary source of evidence that testing has actually been undertaken. If anyone has reason to enquire at a future date about the testing that was undertaken, this is the source that people will access first. The log should be easy to understand and it is fundamental that it provides chronological details of the tests executed. There are various test management tools that provide good test logging facilities e.g. Test Director but if you are not using a test management tool, a simple test log should be constructed along the following lines:

The Test Log

Phase / Stage:			
Test Cycle : Test Date :			
Date	Script Description Details	Result (Pass/Fail)	Comments
Completed by: Date:			

There are many fields that may be found on the test log. There may be a record of who ran the test, how long it took to run, what Cost Code or Cost Centre it should be recorded against etc.

The Tests

The actual test cases that are planned to be run need to be documented. This is achieved by writing test scripts, which record the details of the test (including full instructions of how to perform the test), the input data and the expected results.

Example Test Script

Test Script <Name>				
Seq No.	Detailed Instructions	Test Criteria	Expected Results	Pass/Fail
Tested by: Date:				

1.4.5. Test closure activities

When the testing is complete - all completion criteria have been met - the testing project can be closed. All necessary documents should be updated and put under version control.

Test closure activities collect data from completed test activities to consolidate experience, testware, facts and numbers.

The following tasks are involved in the test closure activities:

- Checking which planned deliverables have been delivered, the closure of incident reports or raising of change records for any that remain open, and the documentation of the acceptance of the system.
- Finalizing and archiving testware, the test environment and the test infrastructure for later reuse.
- Handover of testware to the maintenance organization.
- Analyzing lessons learned for future releases and projects, and the improvement of test maturity.

The exact way that a testing project is closed will depend on the specifics of your organisation.

1.5. The Psychology of Testing

What Makes a Tester?

As the purpose of testing is to find errors it can be defined as being a "destructive" process - development is a "constructive" process as it involves the creation of objects.

The one thing that it is not possible to test for is to show that the software is correct and error free. The purpose of testing is to find faults with the application. This will by its nature, damage confidence in the application.

In order to understand the system, how it works and interacts with other systems, testers must be able to ask various people questions on various topics.

Often incomplete information is available to the testers and they have to work with what is available. Again this is reflected when actually testing elements of the application may be gradually made available to test.

A good tester has these qualities:

- methodical and systematic.
- tactful and diplomatic (but firm when necessary).
- sceptical, especially about assumptions, and wants to see concrete evidence.
- able to notice and pursue odd details.
- good written and verbal skills (for explaining faults clearly and concisely).
- very good in anticipating what others are likely to misunderstand. (This is useful both in finding faults and writing bug reports.)
- a willingness to get one's hands dirty, to experiment, to try something to see what happens.
- able to swiftly understand the nature of the business (so that they can test the systems business usage as well as basic functionality).

In addition to understanding the system, testers must also understand the environment (hardware, infrastructure, operating systems, software) surrounding the system and any likely impact it may cause to the system.

Additionally, a broad understanding of IT issues, technological concepts and the commercial aspects and impact of IT will enable the tester to better relate to the customers' situation.

Additionally testers need to be able to:

- Know how to find faults - how to plan, prepare and execute tests in order to do so.
- Know how to understand the customers IT systems, their purpose and usage.
- Know how to read a system specification to understand the purpose and structure of the system.
- Know how to look at a system and its specification and from these understand the areas to be tested.

Not Just Showing it Works

Testers need to be able to read the specification of a system and devise tests that will ensure that the system works according to the specification. However, they also need to be able to "step outside" the specification and devise tests that ensure the system will reject invalid inputs. By its nature a system specification is a positive document, it defines *what will be* accepted by the system (they do not normally detail *what won't be* accepted).

Testing to Find Faults

Creating tests that prove that the system works as expected (i.e. accepts values that are within the range defined in the specification) are known as valid tests (i.e. the value is a valid amount). These are also known as positive or clean tests.

Creating test cases that use values outside of the valid ranges are known as invalid tests. These are intended to show that the system will reject an invalid value when input. These are also known as negative or dirty tests.

Reporting Defects

Once defects have been found they need to be communicated to both developers and management:

- Development need to understand the fault to be able to provide a fix. They need to know how the system has deviated from the specification and the expected results so that they are able to restore the software to its specification.
- Management need to understand the impact of faults upon testing and development and the risk of the fault to the business.

Communication with Developers

Testers need to develop good working relationships with the developers and be able to report defects back to them.

- They must provide the developers with sufficient information for them to be able to fix the defects.
- They must be able to highlight the defects to the developers without risk of the developers taking offence because they have been found to have made a mistake.
- Testers rely upon the developers advising them that the fault has been fixed so that the system can be re-tested to ensure the fault is fixed.
- When testers find faults that are difficult to replicate a good relationship is essential.

It is necessary to be careful when feeding back faults to developers. Being aggressive and telling a developer that they have made a mistake will result in the developer becoming defensive and perhaps blaming the fault on the testers' lack of understanding. In order to stand the greatest chance of success with a developer it is necessary to provide him / her with sufficient information to find and reproduce the fault. Additionally, a more cautious initial approach - i.e. saying to the developer that you don't understand something and can they help explain it to you - may prove to be a more effective approach.

Communication with Management

Managers need to know:

- How much of the system has been tested.
- The number of faults found and their status (not fixed, fixed & not re-tested or fixed and re-tested).
- The impact of faults upon the testing process and upon development.
- The risk to the business of untested areas and faults.

Management make decisions based on facts! Using metrics is a clear and qualitative way of presenting progress. The key thing that management will be interested in is the risk to the business and the time (and cost) required to fix faults and reduce the risk.

A tester can help management understand and interpret these metrics and then use the EVIDENCE as the basis for the decisions they make.

Testers can also offer advice to help the decision making process, always back any advice given with evidence and don't rely on hearsay, gossip or the grapevine.

However, there are several ways to improve communication and relationships between testers and others:

- Start with collaboration rather than battles – remind everyone of the common goal of better quality systems.
- Communicate findings on the product in a neutral, fact-focused way without criticizing the person who created it, for example, write objective and factual incident reports and review findings.
- Try to understand how the other person feels and why they react as they do.
- Confirm that the other person has understood what you have said and vice versa.

Testing Independence

Testing should ideally be separate from development:

- Developers may feel that they do not need to test their software as they do not make mistakes.
- Developers may attempt to hide their mistakes.
- Developers who do not enjoy testing will make every effort to avoid it.
- Developers may only test their component to ensure it works, it is unlikely they'd test negatively.
- Developers may only test their own modules - they may not consider the need to test multiple components to see how they interact.

Generally it is believed that objective, independent testing is more effective.

The further away from the developers that tests are created the greater the "independence" of those tests.

It is likely that if the developer produces the tests then assumptions made during development are carried through to testing - the developer can have emotional attachments and may even have a vested interest in faults not being found.

By having an independent body create the tests, it is more likely they are created from the system specification rather than from the code (as the developer is likely to do).

The levels of testing independence are illustrated below:

- Test cases are designed by the person(s) who writes the software under test;
- Test cases are designed by another person(s);
- Test cases are designed by a person(s) from a different section;
- Test cases are designed by a person(s) from a different organisation;
- Test cases are not chosen by a person.

1.6. Code of Ethics

Many professions have ethical standards. In the context of professionalism, ethics are "rules of conduct recognized in respect to a particular class of human actions or a particular group, culture, etc."
Because testers often have access to confidential and privileged information, ethical guidelines can help them use that information appropriately. In addition, testers should use ethical guidelines to choose the best possible behaviors and outcomes for a given situation, given their constraints.

Recognizing the code of ethics for engineers, the ISTQB® states the following state 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

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

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

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

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

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

1.7. Summary

Testing Terminology

Error	A human action that produces an incorrect result.
Defect	A flaw in a component or system that can cause the component or system to fail. A defect may cause a failure.
Failure	Actual deviation from the expected delivery, service or result.
Anomaly	A condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience.
Quality	The degree to which a component, system or process meets specified requirements and user needs and expectations.

	Software quality, besides functionality, includes reliability, usability, efficiency, maintainability and portability.
Quality Assurance	Part of quality management focused on providing confidence that quality requirements will be fulfilled.
Quality Control	An activity performed to check that a product is fit for purpose. Software quality control has two aspects; testing is related to the second: **constructive** (avoid defects) and **analytical** (find defects).

Why Testing is Necessary? Why test?

- To find faults.
- To ensure live failures don't impact costs and profitability.
- To ensure user requirements are met.
- To maintain the organisations reputation.
- To measure risk to the business.
- To increase reliability.
- To check whether software adheres to legal requirements or industry specific standards.

Why do errors occur?

Some reasons why errors occur: people make mistakes, poor communication, assumptions, poor / ambiguous documentation, poor training, and pressure.

Testing and Software Quality

Testing does not in itself improve quality - it is merely a measure of quality.

Quality can be measured by testing the relevant factors such as correctness, reliability, usability, maintainability, reusability, testability, etc.

Errors and Where They Occur

Errors can be introduced in many places during the System Development Life cycle (SDLC).

The Cost of Errors and the Cost Escalation Model

A single failure can cost nothing or a lot (e.g. Venus probe). Software in safety-critical systems can cause death or injury if it fails, so the cost of a failure in such a system may be in human lives.

The cost of correcting a defect can be modelled. The cost of correcting the same defect at later and later stages in the SDLC rises dramatically.

Exhaustive Testing

It is usually impossible to test everything (due to limited time and resources).

Risk Management

Therefore testing is used as a method of risk assessment and must be prioritised for those areas of greatest risk to the business. Apportion time spent accordingly to the degree of risk to the business.

When to stop testing

- When money runs out?
- When exit criteria are fulfilled?
- When there is no time left?
- *Theoretical ideal answer* - Depends on the risk to the business.

It is difficult to determine how much testing is enough.

The Fundamental Test Process

A successful test is one that finds a fault (i.e. it has done its job). This is counter-intuitive, as faults delay progress. Thus a successful test is one that may cause some delay. However, the successful test reveals a fault which, if found later, may be many times more costly to correct – so it is beneficial in the long run.

The fundamental test process revolves around the five activities of:

- **Test Planning and Control**

  The test plan should specify how the test strategy and project test plan apply to the software under test.

- **Test Analysis and Design**

  Test analysis and design is the activity where general testing objectives are transformed into clear test conditions and test designs. The application should be analysed in detail from both a user's perspective, and from a technical stand-point.

- **Test Implementation and Execution**

  - Build Test Cases – Includes steps, transaction data, standing data. Cross reference and classify each test.
  - Identify Expected Results.
  - Prioritize test cases
  - Execute test cases
  - Record the result of the executed tests
  - Repeat test activities if necessary

  The test records for each test case should unambiguously record the identities and versions of the software under test and the test specification. The actual outcome should be recorded. It should be possible to establish that all of the specified testing activities have been carried out by reference to the test records.

- **Evaluating exit criteria and reporting**

  The test records should be checked against the previously specified test completion criteria. This should be done for each test level. If these criteria are not met, the earliest test activity that must be repeated in order to meet the criteria should be identified and the test process should be

restarted from that point. Completion or exit criteria are used to determine when testing (at any test stage) is complete.

- **Test Closure Activities**

  Test closure activities collect data from completed test activities to consolidate experience, testware, facts and numbers.

The Psychology of Testing

Testing is performed with the primary intent of finding faults in the software, rather than of proving correctness. Testing can therefore be perceived as a destructive process. The mindset required to be a tester is different to that of a developer.

There are right and wrong ways of presenting faults to authors or management. It is important to communicate between developer and tester: e.g., changes to the application or menu structures that might affect the tests; or where the developer thinks the code might be buggy; or where there might be difficulty in reproducing reported bugs.

Tester must be able to communicate well with Management. Managers need to know how much of the system has been tested, the number of faults found and their status (not fixed, fixed and not re-tested or fixed and re-tested), the impact of faults upon the testing process and upon development and the risk to the business of untested areas and faults.

Generally it is believed that objective, independent testing is more effective. If the author tests, then assumptions made are carried into testing, people see what they want to see, there can be emotional attachment, and there may be a vested interest in not finding faults.

Testing Independence:

- Test cases are designed by the person(s) who writes the software under test;
- Test cases are designed by other person(s);
- Test cases are designed by a person(s) from a different section;
- Test cases are designed by a person(s) from a different organisation;
- Test cases are not chosen by a person.

Expected Results

The required behaviour of the SUT must be predicted under given conditions to ensure that the requirements are being adhered to.

Expected results = expected outcome = expected behaviour of the SUT.

Must be identified before a test is executed.

The "**oracle assumption**" is that a tester can routinely identify the correct outcome of a test.

For every system there will be an **Oracle** – a thing or person from which the expected results can be gathered e.g. the existing system (for a benchmark), or a specification, or an individual's specialised knowledge, but not the code.

Prioritisation of Tests

There is never enough time or resource to test everything. Tests therefore need prioritising to ensure that the best testing possible can be performed in the given circumstances.

Prioritise using:

- Business criticality (risk to the business if that area is not working).
- Technical criticality (complex areas, rate of change, importance to system).
- Visibility of failures to the customers.
- Priority of requirements.

References

1.1.5. - Black, 2001, Kaner, 2002

1.2. - Beizer, 1990, Black, 2001, Myers, 1979

1.3. - Beizer, 1990, Hetzel, 1988, Myers, 1979

1.4. – Hetzel, 1988

1.4.5. – Black, 2001, Craig, 2002

1.5. – Black,2001, Hetzel, 1988

2. Testing throughout the Software Life Cycle

2.1. Software development 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 numerous models for testing. Which model for testing a project or organization will adopt or use depends on the specifics of the system to be tested, as well as on the size of the project.

Verification

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

i.e. "did we build the system right?"

Validation

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

i.e. "did we build the right system?"

Verification is proving that a product meets the requirements specified during previous activities carried out correctly throughout the development life cycle, while validation checks that the system meets the customer's requirements at the end of the life cycle. It is a proof that the product meets the expectations of the users, and it ensures that the executable system performs as specified. The creation of the test product is much more closely related to validation than to verification. Traditionally, software testing has been considered a validation process i.e. a life cycle phase. After programming is completed, the system is validated or tested to determine its functional and operation performance.

When verification is incorporated into testing, testing occurs throughout the development life cycle. For best results, it is good practice to combine verification with validation in the testing process. Verification includes systematic procedures of review, analysis and testing, employed throughout the SDLC, beginning with the software requirements phase and continuing through the coding phase. Verification ensures the quality of software production and maintenance. In addition, verification imposes such an organised, systematic development practice that the resulting program can be easily understood and evaluated by an independent party.

2.1.1. The V-Model (sequential development model)

The essence of the V-Model is:

- To graphically show the stages of the development life cycle (including the testing stages).
- To graphically show the relationship between each development stage and the related testing stage.
- To demonstrate simply that once a basis for testing has been produced (having been verified) then Planning and Preparation can commence.
- To depict the test stages for which the document is to be a source. We recommend that a V-Model should form part of any test strategy, depicting what stages we are doing and the basis from which we are preparing test cases. (Every testing project will have a slightly different interpretation of the V-Model.)

For every document produced as a trigger or essential requirement to a development stage there should be a test, or test stage geared to test the application meets the requirements defined in that specification.

Although variants of the V-model exist, a common type of V-model uses four test levels,
corresponding to the four development levels.

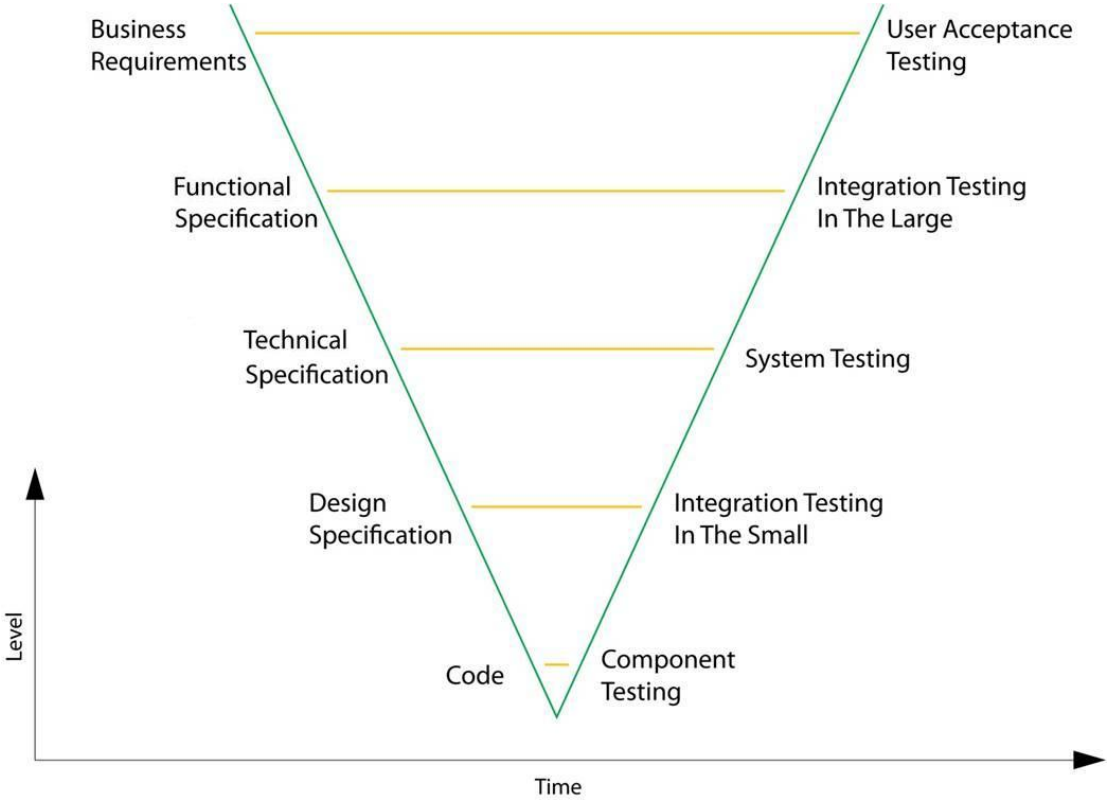
The four levels used in this syllabus are:

o Component (unit) testing
o Integration testing
o System testing
o Acceptance testing

In practice, a V-model may have more, fewer or different levels of development and testing,

depending on the project and the software product. . For example, there may be component integration testing after component testing, and system integration testing after system testing.

Sample V-Model



2.1.2. Iterative-incremental Development Models

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

Incremental development splits the development into a number of increments based on either functionality or some other criteria. Typically, each increment is delivered to the client after it is finished. The model calls for a single requirements phase, followed by a number of increments, each consisting of design, coding, testing and maintenance phases. In principle requirements analysis must be done at the start of the project, but in practice feedback from users can influence both the requirements of latter increments and the design of latter increments.

Examples for iterative-incremental development are: Prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and Agile development. The benefits for testing are:

- Risks are analyzed and eliminated in the early stages of the project. Then revisited in each increment.
- Tight controls of the development process - plans are updated after every increment.

- Special quality increments can be inserted as needed. These usually have "stabilizing" purpose.
- Team motivation is usually higher as the visibility of project status and progress is higher.

A potential pitfall. Project duration seems to increase. This is more of a psychological problem, though. If stakeholders do not have enough understanding of the development process, they may view the increments as slowing down the 'final' product.

2.1.3. Testing within a life cycle model

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

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

Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For example, for the integration of a commercial off the shelf (COTS) software product into a system, the purchaser may perform integration testing at the system level (e.g. integration to the infrastructure and other systems, or system deployment) and acceptance testing (functional and/or non-functional, and user and/or operational testing).

2.2. Test Levels

2.2.1. Component Testing

What is a Component?

"A minimal software item that can be tested in isolation."

A component can be just a few lines of code or it can be a program containing thousands of lines of code. From a requirements point of view a component is a minimal software item for which a separate specification is available.

What is Component Testing?

"The testing of individual software components."

The purpose of component testing is test a software component to ensure that it functions as per its specification.

Component testing is the first form of testing that is performed on the actual code and it is usually performed immediately after the code has been written. Due to the technical nature of component testing - extensive knowledge of the code is necessary - this is usually done by developers. The best person to perform component testing is the components author - (s)he is most familiar with the code and is therefore best placed to be able to find and fix any faults found.

Because it is only a single component that is being tested, the tester will need some kind of "wrapper" in which to place the component to enable it to function on

its own. In a large development project there will be a number of components that do not have any interface (i.e. the components role is to retrieve data from a database and populate a number of variables with this data). These "wrappers" are known as test harnesses or drivers. These may be written specifically for the component or may be a commercially available tool. The driver will need to be able to input data into the component, retrieve the output data from the module and present this information to the tester.

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

Component testing is also known as

- Unit testing.
- Module testing.
- Program testing.

There is a British Standard for component testing - BS7925-2.

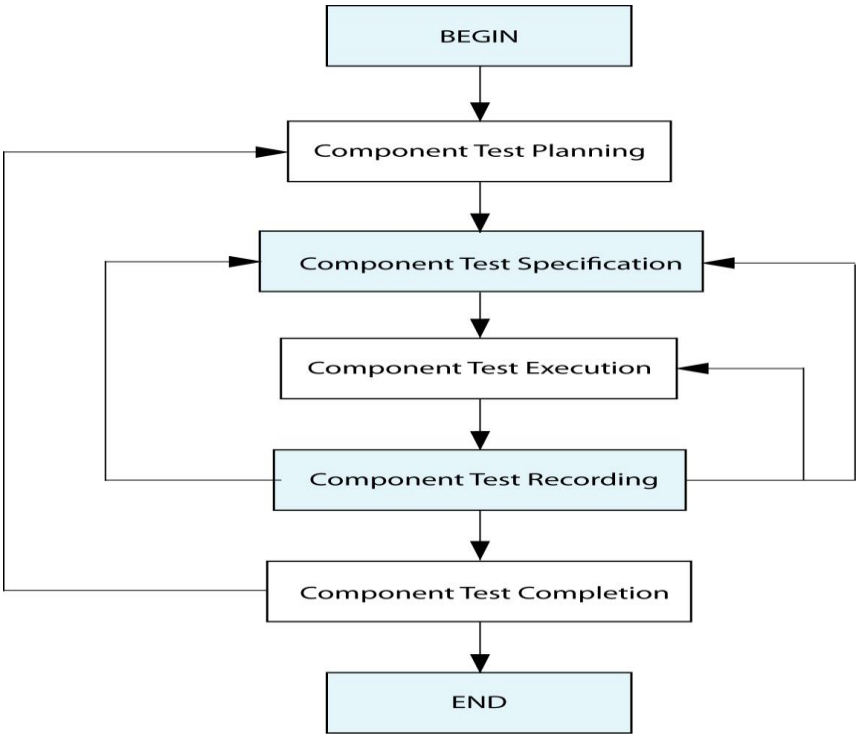
Testing Techniques

The British Standard BS7925-2 defines and details a series of testing techniques and their accompanying measurement techniques that can be used when testing components. The standard covers both Black Box and White Box techniques. *See section Dynamic Testing.*

The Component Test Process

The component test process outlined in the standard is generic and follows the same lines as the test process defined earlier.

Component Test Process



Component Test Planning.

The component test plan is written according to the component (or program) specification. It could be written by the developer – the idea being that it is that person's job to design, build and test the component from start to completion. In other organisations this may not be the case - it may be the analyst's role to perform this task. The test plan should detail the scope of the testing, how the component is to be tested – including the type of tests to be undertaken, and the requirements of the environment.

Component Test Specification

This is the detail of the tests. The detail will be at the code level and will be concerned with the data decision points within the component and/or the paths taken by the data as it is processed. This is one instance of how code coverage is measured. The number of paths through the component is calculated and the data is used to exercise the paths. A monitoring program checks to see how many of the paths have been exercised and reports on the progress. Many test runs may be required to complete the testing, if possible at all.

The specification will also provide the detailed data requirements for each test and the expected result. The time taken to write the Component Test Specification will depend on the size of the component and has been known to vary from a few hours to a few weeks!!

Once the specification is complete then the Plan, Specification and the Code may all be reviewed before test execution begins. The idea of this is to reduce the time spent in laboriously testing the component - desk check removes many defects before the code is exercised.

Component Test Execution and Test Recording

Component Testing is a very iterative process and tends to begin as individual tests. I.e. a single record or piece of information is used to exercise a specific piece of code. The result of this is checked against the expected result and the actual result is recorded as passed or failed.

Depending on the function of the module it may not be possible to test any further than the first test or tests and the developer will have to abandon the testing and fix the component. The fix may also cause the plan and/or the specification to be changed before the testing can restart. This may also require another review of the documentation and the code as well! Once this is ready a retest can be attempted.

As the developer tests further and further into a component it is possible to execute more and more of the tests, testing as much as possible before fixing the faults found. For each round of fixes more changes may be made to the documentation as well as the code and more reviews may be required.

The developer may also try additional tests to ensure that the component will:

- match the interface requirements of any prior or subsequent interface requirements
- cater for blank records without failure and/or providing a sensible message or return code
- handle correctly any field range boundaries
- handle file record sequences correctly e.g. header and trailer processing
- handle database errors: missing data, corrupt database, other invalid return codes etc.

Component Test Completion

Component tests are judged to be complete in many ways:

- All the planned tests have been run
- All the test results have been examined and reviewed to ensure completed correctly
- Time was limited so only the prioritised tests have been run and completed correctly

The most adequate would be for all the tests to be completed, including any re-tests required with a review of the tests undertaken and their results. This review may recommend further tests to take place – in an area say of especial complexity – before the component is finally signed off. A load or stress test may also be considered if there is a concern about the component's performance depending on the position of that component in the overall architecture.

2.2.2. Integration testing

This section examines how we test a system to see how it interacts with the other systems / applications it will be connected to, once deployed. It looks at how applications make up a system, how systems combine to support a full business process and how these systems interact with each other.

What is Integration Testing?

Integration testing helps us understand how data elements are combined to support or drive a business process.

Integration testing

"Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems."

When start integration testing the following basis should be taken in mind:

- Software and system design
- Architecture
- Workflows
- Use cases

Typical test objects in integration testing are:

- Sub-systems database implementation
- Database Implementation
- Infrastructure
- Interfaces

2.2.2.1 Component integration testing

The software development process is invariably based on a modular approach; writing modules that link together to form a complete application of hundreds or thousands of components that interact together. This has proved to be the soundest way of building better quality software applications.

Once a component has been tested on its' own it should be linked to other components and the interaction between them tested.

Integration testing in the small = component integration testing

> *"Testing performed to expose defects in the interfaces and interaction between integrated components."*

Throughout the development of a software application we should consider what internal and external elements are involved in the process or what may be impacted by the introduction of a new application into the workplace.

The main aspects of component integration testing are:

- Once all components have been unit tested they are combined.
- The combination of components is then tested - Integration Testing (in the Small).
- Aim to prove that the data is passed between components as required.
- Gradually linking more and more components together.

The purpose of integration testing in the small is to determine how well the various components that are going to make up a system work together.

Thus, having performed unit testing on the components, we might start by linking two components together. Gradually more and more components are linked together and the tests are expanded.

As more and more components are linked together we begin to form sub-systems. Each sub-system can be tested in isolation and then linked to other sub-systems and tested again. Ultimately, all the components will be combined to form a complete system test.

Test harnesses and stubs will be used to replicate the missing items (components not yet included in the tests or external systems).

Often, when small-scale Integration Testing of several modules or components is performed, it is necessary to devise or improvise methods and tools to get the test data to the components under test. This is often called a test harness. Because of the need to understand the technicalities required to build a test harness this testing is almost always done by the development team.

Obviously there may be faults raised or missed because of deficiencies in the test harness. Therefore more structured testing of these components in a larger scale needs to take place at a later point in the development cycle, when there is a clear pathway to load data - using the applications components - and test it.

2.2.2.2. System integration testing

Integration testing in the large = System integration testing

Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

Integration Testing in the Large is testing performed to expose faults in the interfaces and in the interaction between systems. Integration testing helps with the identification and testing of risks associated with the interaction of "other systems" with the SUT. Once a system has been tested to prove that it functions correctly it is necessary to test it "in situ" i.e. with other systems. Just because a system works correctly when tested in a test environment does not mean it will work correctly (or at all) in the live / production environment.

Integration testing is used to check that the data introduced into the application from other applications is manipulated correctly and that the data passing from the application being tested to other applications is correct for use by them.

Additionally, it is necessary to ensure that the introduction of a new application into the workplace needs to be tested to ensure that it has no adverse impact.

All components of a computer system work on only one thing - data, data is manipulated within the SUT, it may then be passed to another (external) system and / or stored in a database. All this is done to satisfy a business process requirement.

Throughout the development of a software application we should consider what internal and external elements are involved in the process and what may be the consequences of introducing a new application into the workplace.

Integration testing in the large gets increasingly complex as we cross platforms and communication protocols. Consider the additional layers of technology involved with e-commerce and telecommunications systems.

Integration Testing on a Large Scale.

- Test the way the SUT interfaces with external systems.
- The elements may run on the same and / or different platforms.
- They may be located in separate locations and use communication protocols to pass data back and forth.

For applications that are developed to run on more than one platform - such as UNIX, Windows 98 or NT - then the integration testing needs to be performed on each of the target platforms.

Similarly when testing Internet applications, it is necessary to test using the access means (Internet Browsers like Netscape and Internet Explorer) that can be employed to view the web sites.

This therefore may mean testing under more than one operating system or GUI interface with more than one Browser. Worse, you may have to test all existing versions of both the OS and the Browser.

Web and intranet applications will typically have additional layers between the application servers and the user. These include such devices as: load balancers, content scanners and firewalls.

Additional Challenges Posed by Large Scale

- Multiple Platforms.
- Communications between platforms.
- Management of the environments.
- Coordination of changes.
- Different technical skills required.

Highlight where the new challenges lie and that most of these risks can be successfully managed and minimised by having robust plans and management processes and procedures in place.

The more complex or variable that applications become the greater the need to establish the need for continuous testing.

When systems are integrated, they will predominantly integrate with each other in one of the following two ways:

Where the output of system A is read directly into system B

One of the major considerations in integration testing is the location and ownership of the other system (system B), with which you are integrating.

If a third party owns system B (the system with which you are trying to integrate), then the integration test for system A would concentrate on producing the correct output file.

If system A is the external system and system B is internal, then the integration test would be to see if system B could read and accept output files from system A.

However, if both systems A and B were internal systems, then the integration test would involve a combination of the previous two test cases.

Where the output of system A is converted before being read by system B

This particular case is much more involved than those outlined above, as it now involves testing the functioning of a conversion routine. Integration tests in this case would include getting the raw data from system A into the conversion routing, checking the conversion and then getting the converted data into system B.

2.2.2.3. Planning Integration Testing

Planning of testing is the paramount activity. For Integration Testing we have several points in the development life cycle when the test manager needs to consider if integration testing is appropriate and if so what to include - especially for testing in the large.

- We need to determine when and at what levels Integration Testing will be performed.
- Are there any boundaries left?

- A similar process will need to be followed for all elements.
- Once these elements have been tested individually they are further integrated to support the business process.

This is a key point.

We need to think about test environment requirements and the costs of setting them up, populating them with data, managing them and refreshing where necessary. For a large organisation with a variety of applications ranging from desktop to mainframe creation of a test bed to mimic this environment is a major undertaking.

When planning to test sets of components and sub-systems, it is necessary to consider whether one of stubs / harnesses will be required. It is possible that other parts of the system (components, sub systems or external interfaces) will not be available (as they haven't been written) when testing is to start.

Typical approach:

Test the system.

- In a test environment.
- Use stubs where external systems aren't available.

Test the system "in situ".

- In a replica of the production / live environment.
- Use stubs where external systems aren't available.

Test the system and its interfaces with other systems.

- In a replica of the production / live environment.
- Access test versions of the external systems.

Firstly it is necessary to test the system.

The next step is to install the system in a copy of the live environment. This enables the testers to observe the performance of the system installed in a version of the live environment.

Then test the system connected to all the other systems that it will interact / interface with once deployed. Prior to this, the communication with external items will typically have been replicated by a harness / series of harnesses.

2.2.2.4. Approaches to Integration Testing

Incremental

- Top down.
- Bottom-up.
- Functional.

Non-incremental

- "Big bang".

Incremental approaches to integration testing enable us to gradually build the system by steadily adding more and more components together. This means that if a test fails we should be able to quickly identify the likely cause. Incremental approaches require significant preparation and time - each increment will take time to be tested and each phase may need its own harnesses etc. to enable the partial system to function.

Top-down Testing

"An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested."

The main advantages of using top-down method to integration are:

- Early detection of design errors
- Easy location of faults
- A working version of the SUT will be available earlier

Bottom-up testing

"An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. See also integration testing."

This is virtually the opposite of the top-down integration method in that it is based on combining the low-level units into progressively larger modules and sub-systems.

Although this method allows thorough testing to be performed early in the SDLC, the disadvantage is that design errors are discovered late in SDLC, some of which might be serious and it also leads to "Big Bang" situations after integration.

Functional testing

"Testing based on an analysis of the specification of the functionality of a component or system."

The system is tested by function - components being introduced as each function is tested.

Functional integration testing is a combination of both top-down and bottom-up. It is a practical compromise between the two other methods, with the most widely used or critical low level units being developed along with the main top-down integration.

This is the method that is most commonly used when testing projects in the real world.

"Big-bang" testing

"A type of integration testing in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages."

Going for a non-incremental approach offers a considerably faster approach to integration testing. All the components are linked together and tested - there is no gradual, structured increase in the number of linked components. This offers the major advantage of speed - we have one single step rather than many and thus a significantly reduced need for harnesses. However, this approach can make finding the cause of faults a major headache - which of the 2000 components we just linked is causing the problem?

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in

testing the communication between the modules, not the functionality of either module. Both functional and structural approaches may be used.

2.2.2.5. Harnesses, Stubs and Drivers

Drivers and stubs are used to replace missing components/systems during the integration testing.

Drivers are tools used to control and operate the software being tested. One of the simplest examples of a driver is a batch file, a simple list of programs or commands that are executed sequentially. In the days of DOS, this was a popular means for testers to execute their test programs. They would create a batch file containing the names of their test programs, start the batch running and then go home. With today's operating systems and programming languages, there are much more sophisticated methods for executing test programs. For example, a complex Perl script can take the place of an old DOS batch file, and the Windows Task Scheduler can execute various test programs at certain times throughout the day.

Suppose that the software you are testing requires large amounts of data to be entered for your test cases. With some hardware modifications and a few software tools, you could replace the keyboard and mouse of the system being tested with an additional computer that acts as a driver. You could write simple programs on this driver computer that automatically generate the appropriate keystrokes and mouse movements to test the software.

You might be thinking, why bother with such a complicated set-up? Why not simply run a program on the first system that sends keystrokes to the software being tested? There are potentially two problems with this:

It is possible that the software or operating system is not multitasking, making it impossible to run another driver program concurrently. Also, by sending keystrokes and mouse movements from an external computer, the test system is non-invasive. If the driver program is running on the same system as the software being tested, it is invasive and may not be considered an acceptable test scenario.

When considering ways to drive the software that you are testing, think of all the possible methods by which your program can be externally controlled. Then find ways to replace that external control with something that will automatically provide test input to it.

Drivers are used to simulate missing components that will call (drive) the components that are to be tested. In simplistic terms, a driver is a replacement of a missing component that is higher in the system hierarchy than the component under test.

Stubs are essentially the opposite of drivers in that they do not control or operate the software being tested; instead they receive or respond to data that the software sends.

For instance, if you are testing software that sends data to a printer, one way to test it is to enter data, print it, and look at the resulting paper printout. That would work, but it is fairly slow, inefficient, and error prone. Could you tell if the output had a single missing pixel or if it was slightly off in colour? If you replaced the printer with another computer that was running stub software able to read and interpret the printer data instead, it could check the test results much more quickly and accurately.

Stubs are frequently used when software needs to communicate with external devices. Often during development these devices are not available or are scarce. A stub allows testing to occur despite not having the hardware and it makes testing more efficient.

Essentially, stubs are replacements of missing components that the parts of the system being tested will call on as part of the test. And drivers, also called test harnesses, are supporting code and data used to provide an environment for testing parts of a system in isolation.

Typically stubs and drivers will be written specifically for the SUT.

One of the main disadvantages to top-down integration testing is that it does tend to maximise the need to create stubs.

2.2.3. System Testing

What is System Testing?

BS7925-1 definition of System Testing:

> *"Process of testing an integrated system to verify that it meets specified requirements."*

Test basis:

- System and software requirement specification
- Use cases
- Functional specification
- Risk analysis reports

Typical test objects:

- System, user and operation manuals
- System configuration and configuration data

System Testing is the testing of the complete system to ensure that it meets its requirements.

System Testing is usually the first point at which the entire system (or at least part of it, sufficient to test significant areas of functionality) is put together. Often system testing is the final stage in integration testing in the small - all modules / sub-systems are put together.

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

There are two forms of system testing - functional and non-functional. Non-functional testing can also be called quality testing or Testing of Quality Attributes.

2.2.3.1. Functional System Testing

BS7925-1 definition of functional specification:

> *"Document that describes in detail the characteristics of the product with regard to its intended capability."*

Functional system testing is the testing of the systems functionality.

There are two approaches to functional system testing:

- from the functional requirements and
- from a business process.

Requirements-Based Testing

Test cases are derived from Requirements specifications - one of the sources of information used to construct the application.

Tests confirm whether the application satisfies those requirements or not. Always be prepared to test out your tests! Allow for time to do this. Confirm your test plans and test cases with the users. Ensure the data you intend to use is correct etc.

Business-Process-Based Testing

Testing based on user defined scenarios - mimic business processes in the tests. Test cases are created by speaking to users and developing test scenarios from user interviews (or workshops). Testing should therefore reflect the actual or anticipated use of the application. Cases constructed in this way are known as "Use Cases".

2.2.3.2. Non-Functional System Testing

> *Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability.*

Users will direct their attention to the applications' ability to support the functions they are interested in (e.g. can they successfully add a new client to their database and assign an insurance policy to that client).

Non-functional testing directs itself towards other aspects of the application that are as important for its successful deployment.

Usability Testing

How easy is the application to use?

Is it intuitive to use? If the main users of the application are currently used to the more basic Office software (e.g. Word and Excel) the application will need to "look and feel" like those packages.

Are the menu options, tab bars and icons presented in a logical manner that will not force the user to continually scan the display for their next short-cut option?

Usability testing often requires "virgin users" to be brought in to test the system. Testers are not ideal for this as they are usually too familiar with the application and sufficiently computer literate to be able to find their way around the system without help.

Storage Testing

The way the application uses the storage available may need to be considered. When will the database need to be expanded or archived?

Can the storage be modified to make more effective and efficient use of storage?

This is particularly significant for large corporations who are involved with capacity planning.

Installability Testing

When you are happy that the system has been tested and is ready for public consumption - how efficient is it to get the product from the testing environment into the live, public domain.

Can it be installed easily? Does it work correctly after installation? Does it work on system configurations that are not the same as the test environment? Does the software uninstall successfully?

Documentation Testing

While documentation is not a software, it works in conjunction with the software and forms part of the finished product. Therefore it should be tested.

If the documentation is not correct or is misleading it can lead users to report shortcomings in the product when it is not necessarily malfunctioning.

Recovery Testing

Deliberately forcing the system to fail to access its recovery capabilities.

Backup / Restore testing should be the first stage before moving onto recovery scenarios – there is no point in trying to recover a system if the restore process doesn't work!

At various points in the process, break the system to access how to restore the data to a state from which restarts are achievable, how much data would a user need to re-enter? How much effort is required to restore all the necessary connections? How long does recovery take?

Companies will need to know how resilient the system is. Is data lost when there is a hitch? How long will it take to restore the system and its databases?

This may involve a short disconnection from the server (unplugging a wire), seeing if the server copes with a power failure (turning it off) or testing to see if the system is capable of being successfully restored from a back-up.

Load Testing

Will the application be capable of processing the enquiries placed upon it by a number of users carrying out their day-to-day tasks within acceptable time scales? Business use is simulated and key response times are measured. Analysis of shortfalls are carried out, the application amended and the assessment repeated.

Stress Testing

Each major component in an application or system is tested to its limits to assess its durability under stress. This helps to identify when the system in live use, approaches its capacity and guides support personnel to prepare for situations when problems are likely to occur. Also, Mean Time to Failure - MTTF is calculated.

Performance Testing

Performance tests will show the performance of the system - how long it takes for a response to a request etc.

This may be particularly significant if the software developers have agreed Service Level Agreements detailing response times for the system.

Volume Testing

Testing the system to determine how it will handle large quantities of data.

2.2.4. Acceptance Testing

Acceptance Testing is one of the last testing activities to be performed - often it is the final one.

The British Standard definition of Acceptance Testing is:

> *"Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the*

acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system."

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

Acceptance testing may occur in more than just a single test level, for example:

- A COTS (commercial off the shelf) software product may be acceptance tested when it is installed or integrated.
- Acceptance testing of the usability of a component may be done during component testing.
- Acceptance testing of a new functional enhancement may come before system testing.

When preparing Acceptance Tests the following basis should be taken in mind:

- User requirements
- System requirements
- Use cases
- Business processes
- Risk analysis reports

The typical test objects in Acceptance testing are:

- Business processes on fully integrated system
- Operational and maintenance processes
- User procedures
- Forms
- Reports

Planning Acceptance Testing

As with all forms of testing, Acceptance Testing must be planned.

One of the major issues with it is the supply and organisation of the resources required to run the tests. In addition to the normal need for PCs, servers, access to the SUT, databases and data, acceptance tests require an environment that is identical to the live / production system and the users themselves. Simply getting a number of experienced users for the tests can be a major challenge!

2.2.4.1. User Acceptance Testing

User Acceptance Testing (UAT) is where users of the final product conduct the tests and ensure that the system meets their requirements and allows the business to work as expected.

The UAT environment should be representative of a normal business environment; however UAT can also cover other aspects of the project life-cycle and not just the end product.

Users should have been involved throughout the software development life-cycle so there should be no surprises during UAT. Any faults found at this stage may not be fixable as this is usually one of the final stages in the development life-cycle.

The purpose of getting users to perform acceptance testing on the SUT is to get them to accept the system before it is implemented. The other is to ensure that the SUT fulfils the required Business Functionality.

The users would have driven the original requirements for the system; therefore they are the ideal people to confirm that the system matches the requirements.

As they are also the people who will be using the system once it is deployed they should confirm that it is suitable for the business.

2.2.4.2. Operational (acceptance) testing

The acceptance of the system by the system administrators, including:

- testing of backup/restore;
- disaster recovery;
- user management;
- maintenance tasks;
- periodic checks of security vulnerabilities.

2.2.4.3 Contract and Regulation Acceptance Testing

Contract and Regulation Acceptance Testing is done by software houses that have signed contracts with customers that detail what the developers will deliver.

Typically, this is broken down into a number of stages. Before the software is accepted for each stage the customer will wish to ensure that the software performs as per their requirements (and the contract). This is Contract Acceptance Testing.

Often there is a financial tie in for each stage - the software house doesn't get paid until the customer accepts.

2.2.4.4. Alpha and Beta (or field) Testing

Software houses developing shrink-wrapped products usually do alpha and beta (or field) testing. They provide a targeted market (usually a known set of existing users) with a pre-release version of the software.

The customer gets to examine the software before it is released and provide feedback (and perhaps influence its development). The software house, on the other hand, gets its software tested by more people, on more configurations of hardware for free (whilst appearing to favour these customers).

Alpha Testing:

> *"Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization.*
>
> *Alpha testing is often employed as a form of internal acceptance testing."*

Beta Testing:

> *"Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes.*
>
> *Beta testing is often employed as a form of external acceptance testing in order to acquire feedback from the market."*

The fundamental difference between alpha and beta testing is where they are carried out. Alpha testing is done on the development site (but outside of the development environment); beta testing is done on the customer's site (and is therefore out of the software houses control).

2.3. Test types

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

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

A model of the software may be developed and/or used in structural and functional testing. For example, in functional testing a process flow model, a state transition model or a plain language specification; and for structural testing a control flow model or menu structure model.

2.3.1. Testing of function (Functional Testing)

The functions that a system, subsystem or component are to perform may be described in work products such as a requirements specification, use cases, or a functional specification, or they may be undocumented. The functions are "what" the system does.

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

Black-box testing techniques may be used to derive test conditions and test cases from the functionality of the software or system. Functional testing considers the external behaviour of the software.

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

Security Testing

This is concerned with ensuring that the system and its databases are secure from unauthorised access. This may involve simply testing passwords or deliberately attempting to hack into the databases.

In certain areas there are very strict data protection laws - Germany for example. In such places there are legal obligations that must be considered when testing these aspects of a system.

2.3.2. Testing of non-functional software characteristics (Non-Functional Testing)

The non-functional aspects of a SUT, which are also known as "quality" aspects include such aspects as security, performance, load, volume, stress, usability, maintainability, etc. This category of testing technique is concerned with examining how well the system does something, not what it does and how it does it.

The techniques used to test these non-functional aspects of a SUT are less procedural and less formalized than those of other categories. This is because the actual tests are more dependent on the type of SUT.

Non-functional testing may be performed at all test levels. The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale, such as response times for performance testing. These tests can be referenced to a quality model such as the one defined in 'Software Engineering – Software Product Quality' (ISO 9126).

2.3.3. Testing of software structure/architecture (structural testing)

Structural testing uses the internal structure of the software to derive the actual test cases. Structural testing techniques are commonly referred to as 'white-box' or 'glass-box' techniques (implying that you can see into the system) since they all require knowledge of how the software is implemented, i.e. how it works. For example, a structural technique may be concerned with exercising all the statements in the software. Different test cases may be derived to exercise the selective or all the statements at once, twice, or even many times.

Therefore, structural testing concentrates on the actual code and its execution rather than the end functionality of the SUT. Knowledge of the code is essential for structural testing.

Structural techniques are best used after specification-based (black-box) techniques, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure.

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

At all test levels, but especially in component testing and component integration testing, tools can be used to measure the code coverage of elements, such as statements or decisions. Structural testing may be based on the architecture of the system, such as a calling hierarchy. Structural testing approaches can also be applied at system, system integration or acceptance testing levels (e.g. to business models or menu structures).

2.3.4. Testing related to changes - Confirmation testing (Re-testing) and Regression Testing

2.3.4.1. Confirmation testing (re-testing)

Confirmation testing and re-testing a two interchangeable terms. Both terms appear in the syllabus, re-testing is the term that is used in the handouts.

The purpose of testing is to find faults. Once a fault has been found it will be reported to the developer so that it may be fixed. Once the fault has been fixed a new version of the software will be released with the fix included.

It is not sufficient for testers to simply accept the developer's word that the fault is no more. The test(s) should be run again to ensure that the fault has indeed been fixed. This is known as re-testing.

Fault Fixing and Re-testing

The test process itself is naturally iterative.

Iterative testing simply means testing that is repeated, or iterated, multiple times. Iterative testing is important because the ultimate goal of all the work being undertaken is to improve the SUT, not just to catalogue the problems.

For instance, a single usability test - particularly if no action is taken based on its findings - can only tell you how successful or unsuccessful you were in creating ease of use. To improve upon what you already have, recommendations based on the usability test's findings must be incorporated into a revision of the product. Once this has been done, it's advisable to test the product again to make sure that no additional usability flaws were incorporated with the fixes to the previously found glitches.

In an ideal world, of course, this cycle of testing would continue as long as meaningful recommendations for improvement could be drawn from the usability test results. In reality, it's best to define quantifiable goals for your product's usability before you begin testing, and to continue the cycle of testing and revising until your usability goals have been met.

Therefore, if a fault is found it must be fixed and retested. This leads to three possible issues:

- Is the test case repeatable? If the test case is not repeatable, the fault cannot be recreated and demonstrated to the developer.
- Is the test case repeatable so that it can be proven that the fix has actually worked?
- Has the fix somehow introduced one or more new faults into the SUT?

Test Repeatability

One of the characteristics of a good test case is that it is easy to repeat. There is nothing worse from a testing perspective than to find a fault and then not to be able to recreate it. Test case repeatability is determined by two crucial factors:

- Having a fixed point to start from. As test environments are constantly changing, it is important from a testing perspective, that the precise input and the state of the system when the input occurred are known.
- Knowing exactly what data or keystrokes were entered into the system and in what sequence.

2.3.4.2. Regression Testing

After the initial faults have been rectified, solving one fault may reveal another 'dormant' defect in the code.

The purpose of regression testing is to determine if the system (and the quality of the system) has "regressed" following a change.

Regression testing is defined as:

> *"Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered as a result of the changes made. It is performed when the software or its environment is changed. "*
>
> A crucial word to notice in this definition is "Uncovered"… this harks back to the point we have been making that exhaustive testing is impossible…

Systems will go into production with faults, these faults may not be found while the application is in production use, and a structured approach to testing will help to flush the problems out before the users find them!

Regression tests should be run whenever there is a change to the software (i.e. a new release) or to the environment (i.e. new database, new server).

Automating Regression Testing

A Regression suite is the ideal foundation on which to build automation.

The tests will be used many times, not only for new releases of software, but for hardware or environment upgrades, and smaller emergency fixes of software.

Regression suites can also be used to continuously test an application, after it has been made live to constantly verify the functionality of an application, particularly if it is running in a dynamic environment, 24 hours a day 365 days a year (e.g. Web-based applications or services).

Planning Tests for Re-Testing and Regression Testing

Testing does not happen once and is then forgotten. Tests may need to be used and re-used many times over. This potential needs to be considered when the tests are being planned and designed.

A test may need to be repeated if at any time it highlights a defect, once the developers have fixed the problem it will need to be re-tested to prove the fix. Other tests in related areas should also be re-run to ensure that fixing one problem has not caused previously working code to malfunction. It is also quite possible that fixing one problem may reveal other, dormant, errors that were not apparent or observable.

The test schedule needs to allow time throughout the testing phase for re-testing and regression testing. In the early days of testing, it is quite possible that the percentage of re-testing is high. It will perhaps be the first time all the components have been built together in a new environment - in development the application would have 'evolved'. It is highly likely that it is the first time the application has been subject to any independent scrutiny.

Each test case should be repeatable as an entity in its own right, as far as it is practical.

Each test case should have a single clear objective.

It may be necessary to have the environment that the tests are run in quickly and efficiently restored to a predefined, known state. This will mean that tests can be run repeatedly and, as the SUT is in the same condition every time, the results from each test will be comparable.

Selecting Regression Test Cases

Whilst it is obvious that we need to test the area in which the change was made (to ensure that the change has been made successfully) we also need to perform a broader series of tests to ensure that the change has not impacted upon any other area of the system.

USE CASES are probably the most effective way to create a suite for the application.

- Find out from the users how they use the application. Plan tests to cover that use.

- Find out from the Business what the major processes in the application are - ensure those are covered with the Use Cases - if not plan more tests!!

There are currently three main methods for selecting regression test cases:

- **Safe methods**

  These guarantee that all test cases that revealed faults are selected and re-run.

- **Coverage-based methods**

  These attempt to find test cases that meet specific coverage criterion

- **Retest-all approach**

  This approach is currently prevalent at the moment and it is where no test case selection at all is undertaken.

2.4. Maintenance Testing

What is Maintenance Testing?

> *"Testing the changes to an operational system or the impact of a changed environment to an operational system."*

This module looks at the challenges that face testers when the applications change. How to ensure that maintenance applied to the system does not cause failures.

Rather like a car - computer applications will need regular services and maintenance. Your car service may reveal that the brake fluid needs changing, this is done but as part of the work you rightly expect to be able to drive off and not find your engine catches fire (the new fluid leaks onto the engine, it ignites…) as well as the fact that you can now brake safely.

Again in the car - the brakes can be testing in-situ on a rolling road. But the ultimate proof is when you come to actually run the car on the road.

After any work on a car is carried out it should be normal practice for your garage to take the car out for a short…… Test Drive… Not necessarily a thrash on the motorway but a quick spin round the block.

In IT we strive to do the same.

Maintenance testing is triggered by modifications, migration, or retirement of the software or system.

Modifications include planned enhancement changes (e.g. release-based), corrective and emergency changes, and changes of environment, such as planned operating system or database upgrades, or patches to newly exposed or discovered vulnerabilities of the operating system.

Maintenance testing for migration (e.g. from one platform to another) should include operational tests of the new environment, as well as of the changed software.

Maintenance testing for the retirement of a system may include the testing of data migration or archiving if long data-retention periods are required.

Testing Changes

When a change to the system is being introduced, it must be tested both in isolation i.e. prior to being integrated with the current system and as part of the

system, once integration has taken place. If the proposed change is being tested in isolation, it is likely that stubs and drivers would probably be used to create the framework or harness to test it within. When the change is subsequently incorporated into the full system, a regression test pack must be run to ensure that no new problems have been introduced and no existing problems have been uncovered, as a result of the change that has taken place with the system.

The use of impact analysis is very helpful in this instance because if the functional hierarchy is cross-referenced back to the requirements, then for any change in the documentation, it is an easy exercise to find what functionality maybe impacted and where it is tested. This will save a lot of time in selecting test scripts to amend and re-run.

Maintenance may be needed to bring applications in line with changes to regulations. Old applications, untouched for years need to be maintained.

The system will then need to be tested to ensure that the maintenance has been done successfully. It will also need to be tested to check that the maintenance has not damaged any other part of the application or brought to light any other existing faults.

Hence, maintenance testing is very much focused on regression testing – ensuring that the quality of the system is unaffected by the maintenance.

Risks of Changes

There are four results that can occur from a change being made to the software:

- The change itself was successful but a new fault(s) was introduced
- The change itself was successful and no new fault(s) was introduced
- The change itself was unsuccessful and no new fault(s) was introduced
- The change itself was unsuccessful and a new fault(s) was introduced

Only the second outcome listed above is acceptable. Any new faults that are introduced during changes that are made to the system, e.g. fixing faults, are much more difficult to find and fix than others.

The Challenges of Maintenance Testing

Because maintenance testing involves systems that are in place and are in use (and perhaps have been for years) it has its own set of challenges.

- The documentation may have gone missing (or is bad or, worse, is out of date).
- How do we know what to test?
- What are the relationships within the application?
- What's important to test?

With a system that is in place and is being used by the business, there is a need to perform impact analysis, in order to understand the risks to the business of the system not working when the new version is deployed.

There are other risks to consider when carrying out maintenance…

- If the software is installed into production with too little testing other errors may manifest themselves, and cause another outage.

- Confidence will be eroded, business lost and more time expended on solving & then re-testing the system.
- If too much testing is carried out before the system is restored potential business may have gone elsewhere and revenue lost.

2.5. Summary

Models for Testing

Verification – determining whether the software products of an activity fulfil the requirements or conditions imposed on them in the previous stage. "Did we build the system right?"

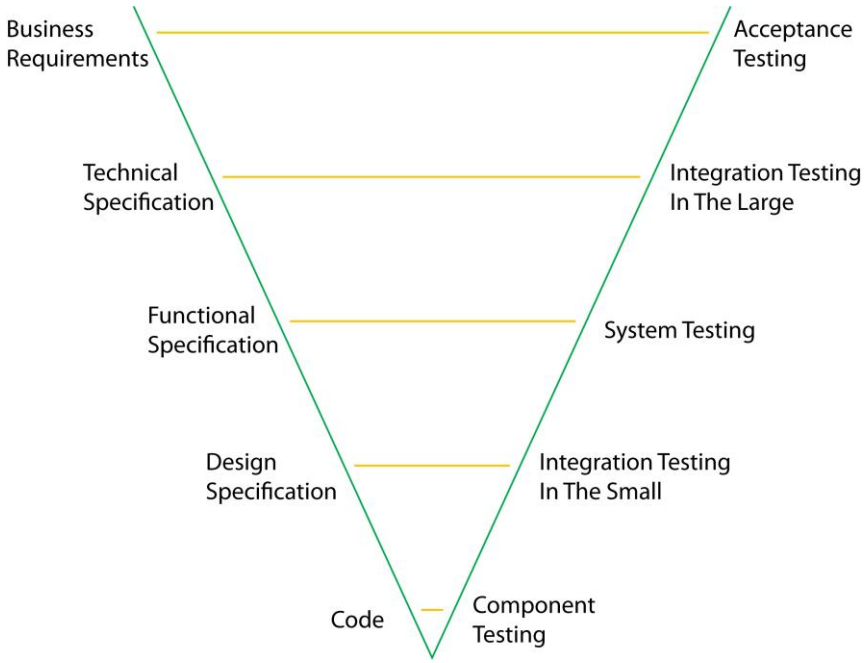
Validation – determining whether the requirements and the final, as-built system or software product fulfils it specific intended use. "Did we build the right system?"

V-Model

The essence of the V-Model is:

- To graphically show the stages of the development Life-cycle (including the testing stages).
- To graphically show the relationship between each development stage and the related testing stage.
- To demonstrate simply that once a basis for testing has been produced (having been verified) then Planning and Preparation can commence.

Economics of Testing



The cost of faults escalates as we move the product towards field use. Early test design can prevent fault multiplication. Analysis of specifications during test preparation often brings faults in specifications to light.

The cost of testing is generally lower than the cost associated with major faults (such as poor quality product and/or fixing faults), although few organisations have figures to confirm this.

Component Testing

Component-based testing is also known as Unit, Module, or Program testing

A component is a *minimal software item that can be tested in isolation.*

Component Test Process:

- Component Test Planning
- Component Test Specification
- Component Test Execution
- Component Test Recording
- Checking for Component Test Completion

Component Integration Testing (Integration Testing in the Small)

Assembling components into sub-systems and then sub-systems to systems.

Integration testing tests interfaces and interaction of modules.

Integration testing in the small is the testing of the components of a system and their interface & interaction with other of the system's components.

Because integration testing in the small is often done in parallel to development and thus parts of the system will not be available we need to emulate these missing parts. This is done with a mixture of stubs and drivers.

- Stubs are replacements for missing components that the components being tested will call as part of the test.
- Harnesses and drivers are used to simulate missing components that will call (drive) the components that are to be tested.

Typically stubs and drivers will be written specifically for the SUT.

Incremental integration strategies:

- Top-down – start with the highest level components and gradually add lower level components.
- Bottom-up – start with the lowest level components and gradually add higher level components.
- Functional – add components function by function.

Non-incremental integration approach:

- "Big-Bang" – add all the components at once.

Functional System Testing

Functional system testing is the testing of the systems functionality.

There are two approaches to identifying and building functional tests:

Requirements based

Tests are derived from the requirements specification.

Business Process based

Tests are based on user defined scenarios and should therefore mimic business processes.

Non-Functional System Testing

Non-functional testing is concerned with the areas of the system that do not relate to business functionality. These will be things such as how does it look and feel and what is its performance like.

There are a number of different types of non-functional test types:

- **Security** – This is concerned with ensuring that the system and its databases are secure from unauthorised access.
- **Usability** – How easy is the application to use?
- **Storage** – The way the application uses the storage available may need to be considered.
- **Installability** – Can it be installed easily?
- **Documentation** – Is the system documentation of sufficient quality and is it accurate?
- **(Disaster) Recovery** – Deliberately forcing the system to fail to assess its recovery capabilities.
- **Load** – Will the application be capable of processing the enquiries placed upon it by a number of users carrying out their day to day tasks within acceptable time scales?
- **Performance** – Performance tests will show the performance of the system - how long it takes for a response to a request.
- **Stress** – Each major component in an application or system is tested to its limits to assess its durability under stress.
- **Volume** – Testing the system to determine how it will handle large quantities of data.

System Integration Testing (Integration Testing in the Large)

Integration Testing in the Large is testing performed to expose faults in the interfaces and in the interaction between systems, the integration with other (complete) systems. Identification of, and risk associated with, interfaces to these other systems.

Additional challenges posed by large-scale integration testing:

- Multiple Platforms.
- Communications between platforms.
- Management of the environments.
- Coordination of changes.
- Different technical skills required.

Acceptance Testing

Acceptance testing is

> *"Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system."*

- User Acceptance Testing (UAT) – Performed by the 'end users' to ensure that the individual elements (menu item, screen, report) and the overall system, provide the desired result or functionality.
- Operational (acceptance testing) - the acceptance of the system by the system administrators.
- Contract acceptance testing – A demonstration of the acceptance criteria, which would have been defined in the contract, being met.
- Alpha testing – People who represent your market use the product in the same way(s) that they would if they bought the finished version and give you their comments. Performed at the developer's site.
- Beta testing – As per Alpha testing, but tests are performed at the users' site.

Test types:

- Testing of function (functional testing)
- Testing of software product characteristics (non-functional testing)
- Testing of software structure/architecture (structural testing)
- Testing related to changes (confirmation testing and regression testing)

Maintenance Testing

Testing old code (i.e. established applications that have been in place and used for years) that has had maintenance done (i.e. changes made).

Usually with poor / missing specifications i.e. established application that may have been installed and working for years.

Impact analysis is vital and difficult –higher risk when making changes

Regression testing is a must, but difficult to decide how much to do (i.e. when to stop it - do you stop early and risk missing faults or do you stop late and damage business?).

References

2.1.3. – CMMI, Creig, 2002, Hetzel, 1988, IEEE 12207

2.2. – Hetzel, 1988

2.2.4. – Copeland, 2004, Myers, 1979

2.3.1. – Beizer,1990, Black, 2001, Copeland, 2004

2.3.2. – Black, 2001, ISO 9126

2.3.3. – Beizer, 1990, Copeland, 2004, Hetzel, 1988

2.3.4. – Hetzel, 1988, IEEE 829

2.4. – Black, 2001, Craig, 2002, Hetzel, 1988, IEEE 829

3. Static techniques

3.1. Static techniques and the Test Process

What is Static Testing?

> *"Testing of a component or system at specification or implementation level without execution of that software, e.g. reviews or static code analysis."*

Question: Why would we want to test an object without executing its code?

Answer: Because we don't have the code yet or are unable to execute it. Or because the code has been written and compiled ready to run and is to be desk-checked against the specification first before test execution starts

Static testing involves a person or people examining an object (documentation & product deliverables). By reviewing these items we should be able to find and fix faults early on in the software development life cycle. Another reason this approach was adopted was that it enables code to be reviewed without the need for expensive machine time.

As soon as an object is ready it can be reviewed, (although code can more or less be reviewed at any time, depending on the skill of the developer). This would be before it is used as a product or the basis for the next step in development.

Reviews can be performed well before dynamic test execution. Defects detected during reviews early in the life cycle are often much cheaper to remove than those detected while running tests (e.g. defects found in requirements).

What to Review?

- Anything and everything can be reviewed.
- Requirements, System and Program Specifications should be reviewed prior to publication.
- System Design deliverables should be reviewed both in terms of functionality and technical robustness.
- Code can be reviewed.

During the SDLC, a myriad of various deliverables are produced. Each of them contributes to the delivered application. Therefore if any one is erroneous or has significant omissions then the application will be defective.

As well as testing code for defects, it should ensure that the deliverables that have been produced are subject to some kind of review to assure that:

- It is fit for purpose
- It is complete
- It meets all its objectives
- It is fit to be input to the next stage of the Process

Many ways exist to review deliverables to ensure that the goals above are met.

NOTE: This does not say that the review must be carried out by the testers - it is the project managers' responsibility to assign the task to the correct person / people. Many companies have a Quality Assurance department that performs this role.

What Should be Reviewed?

- Program specifications should be reviewed before construction.
- Code should be reviewed before execution.
- Test Plans should be reviewed before creating tests.
- Test Results should be reviewed before feature implementation in a new build.
- All development and design documentation.
- Test cases.
- Causes of defect.
- Test Metrics, Development Metrics.
- Operations defects, Defects.

The role of testing is to find and eliminate defects. We want to ensure that when the application is implemented and goes into live operation it is as good as it needs to be to serve the business needs and objectives.

There is no reason to wait until code is ready to start testing and testing-related activities. Once the Requirements have been specified then the work of deriving User Acceptance Test Cases can commence.

Before this the Requirements document(s) should be reviewed:

- To eliminate ambiguity that leads to the wrong product being built.
- To identify and correct all errors.
- To ensure that all areas are fully specified with no omissions.

The Benefits of Reviews

- Reviews save time and money.
- People take greater care.
- They have more pride in doing good work.
- They have a better understanding of what they are required to deliver.
- Remove defects / errors earlier.

An organisation that implements a strong review culture soon finds that the quality of their deliverables improves greatly - both before and as a result of the Review. This is because the whole ethos of the people in the organisation becomes one where each individual accepts responsibility for his or her own work. They naturally feel that if someone else is going to assess their work that they need to ensure it is of assessable quality.

Because QA and review processes have to be defined, many people have to contribute to the process definition and everyone has to buy into it.

Anachronistically, this extra activity actually reduces overall development time. This is due to the higher accuracy and lower defect rate giving rise to far less re-work. And that the re-work takes place earlier in the SDLC - which obviously lessens the time and cost of defect repair.

Systems development is very complex. This means that it is hard to build a system with all elements completely & accurately specified.

By reviewing a deliverable, its deficiencies can be identified and remedied earlier – saving **TIME and MONEY!**

It has been estimated that a formal review process implemented throughout the SDLC may require additional effort of between 10 - 15%. This includes the production of metrics and some remedial activities.

This should be more than compensated for by the much reduced amount of re-work discovered later in the SDLC - let alone during live running.

Over time, maintenance of live systems is, by far, the largest expense. Therefore by developing systems that subsequently require far less change will be a major factor in reducing over system costs.

There is a cost associated with reviews - the time involved in getting people together. A typical cost of an on-going review process is 15% of the development budget. The cost of reviews includes the review process itself plus metrics analysis and process improvements.

The Risks of Reviews

- If misused, they can lead to friction.
- The errors and omissions found should be regarded as a good thing.
- The author(s) should not take errors and omissions personally.
- It is a positive step to identify defects and issues before the next stage.

It is basic human nature to take pride in ones work. Similarly it is human to be protective of it and hurt when it is criticised. For reviews to be successful a mature approach is necessary. People must acknowledge that, being human we will not get any document of significant complexity 100% right. Once this is accepted by all involved, then it is easy to accept that others can build on ones work to make it better.

If one is introducing reviews to an organisation, everyone needs to understand and commit to the no blame attitude of all review processes.

There are other dangers in a regime of regular reviews:

- Lack of Preparation.
- "Familiarity breeds contempt".
- No follow up to ensure correction has been made.
- The wrong people always doing them.
- Used as witch-hunts when things are going wrong.

All reviews require some preparation. Even the author when reading his / her own work should make a list of what (s)he is checking for.

Before you ask someone else to read it ask if they have enough time to do so.

In a formal review or inspection all participants should have reviewed the document and made a list of defects and comments. **If this is not done, there is no point in continuing the review.**

Over time "lip service" culture may well creep in. People may cut corners and not prepare properly. Key staff will excuse themselves from reviews as being too busy and may delegate to others without the requisite skills or knowledge.

Once complete, perhaps nobody will ensure that the corrective actions identified at the review have been taken.

Building a Quality Culture

In order for them to work, Reviews must be regarded as a positive step and must be properly organised. This is often a part of the company's Quality culture.

In order for reviews to be successful it is necessary to adopt a formal approach, this will typically include the following steps:

- Define the review panel.
- Define the roles of its members.

- Define the review formal procedures and structures.
- "Sell" the quality culture to staff.
- Company culture is enhanced to include Quality Control.

Additionally, they may include a recognised Quality certification - ISO 9000.

What Else Can Testers Review?

- All development and design documentation (It's where the test cases come from!!)
- Test plans.
- Test cases.
- Test results.
- Defects.
- Causes of defect.
- Test metrics.
- Development metrics.
- Operational defects.

We can measure how well we do

As testers it is important that we also ensure that the deliverables we produce are as good as they need to be. The same techniques used to assure quality of Development deliverables are used to ensure that the test assets we build are fit for purpose.

We should review all our deliverables ourselves. Moreover we should publish our test plans and test documents widely, to ensure that all aspects of the testing have been considered by Development and the Business.

Causal Analysis of defects is a powerful mechanism to identify defects in the SDLC *and the Test Process*. From such an analysis, Management can identify weaknesses in the process and identify appropriate remedial actions.

Causal analysis can be done at two levels. It can be done by examining Production faults to see where the errors are caused and chase the fault back to the root cause, or by examining the faults found while testing or reviewing in order to see again where the problems have arisen. Remedial actions can be put in place to ensure they don't recur.

In good quality systems training is targeted to ensure that weaknesses are eliminated, other types of errors are frankly and openly discussed in an open forum. This requires a no-blame culture, if there is a fault then that fact is accepted and everybody must pull together to fix it.

Reviews, static analysis and dynamic testing have the same objective – identifying defects. They are complementary: the different techniques can find different types of defect effectively and efficiently. In contrast to dynamic testing, reviews find defects rather than failures.

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

Summary

- Reviews enable us to test that the systems specification is correct and relates to the users' requirements.

- Reviewing specifications gives testers the opportunity to find faults in the specifications (i.e. discrepancies in the business process) before coding has started.
- Anything generated by a project can be reviewed.
- In order for them to be effective, reviews must be well managed.
- The company must have a no-blame culture for them to succeed.

3.2. Review process

3.2.1. Phases of a formal review:

- **Planning**: Selection of the personnel group, allocation of roles, determination of the maturity of the object to be reviewed,
- **Defining the entry and exit criteria for more formal review types (e.g., inspections)** - selecting which parts of documents to review
- **Kick-off**: Object for review and accompanying documents such as guidelines, reference documents, test criteria etc. to be available on time, explaining the objectives, process and documents to the participants; and checking entry criteria (for more formal review types),
- **Checking entry criteria** (for more formal review types)
- **Individual preparation**: individual preparation of the participants for the review meeting, noting recognized deficiencies, questions and comments,
- **Noting potential defects, questions and comments**
- **Examination/evaluating/ recording of results (Review meeting):** discussion or logging, with documented results or minutes (for more formal review types). The meeting participants may simply note defects, make recommendations for handling the defects, or make decisions about the defects. **Examining/evaluating and recording during any physical meetings or tracking any group electronic communications**
- **Rework**: fixing defects found, typically done by the author.**Fixing defects found (typically done by the author)** - recording updated status of defects (in formal reviews)
- **Follow-up**: Arrangements for a follow-up event, recognition and curing deficiencies in the development process; checking that defects have been addressed, gathering metrics and checking on exit criteria (for more formal review types)
- **Checking on exit criteria (for more formal review types)**

The normal goal of a review process is to perform verification and validation on the reviewed item - to check it for faults and to ensure that it conforms to its specification / requirements.

A number of the review types (prototyping and presentations) involve explaining a concept or idea to a number of people - typically the end users of the system being developed. For these sessions the key goal is to gain consensus for the idea(s) from the attendees.

Deliverables

The main deliverable from a review is a list of changes to be made to the item being reviewed. Additionally there may be a further list detailing a number of

queries raised by the reviewers about the item - these will need to be investigated and responded to by the author outside of the review itself.

Pitfalls

If correctly run reviews can prove very successful in finding faults in a system. However, they can also be a complete waste of time and actually hinder the software development process.

For reviews to stand a chance of success:

- They need to be correctly managed.
- Attendees need to have a clear understanding of their role and responsibilities.
- They may need training to partake in the review.
- Both management and review staff need to support the process.

Perhaps the most challenging problem with reviews is the motivation of the staff involved in them. Some of the issues with reviews are as follows:

- They are perceived to be a waste of time because the quality of the documents being reviewed is poor.
- The time could be better spent on other things - such as actually writing code.
- They delay the development process.
- No time is allowed for reviews therefore no more than "lip service" is paid to them.
- They solve nothing - often because there is no follow-up process in place to ensure that corrections have been made or queries resolved.

3.2.2. Roles and Responsibilities

Within any of the formal review procedures it is necessary that all attendees understand their role and responsibilities in the review. The review group should comprise a diverse group of people and each person should have their own unique role to play within the group.

- **Manager.** Decides on the execution of a review. Responsible also to promote and encourage the review process and to be open to any new ideas that might flow from the review.
- **Moderator.** Leads the review as a neutral person. Facilitates the review process and ensures that the review stays focused on the goals identified at the outset. If necessary, mediates between the various points of view and is often the person on whom the success of the review rests. This is usually a specially trained person.
- **Author.** This is the writer or person with chief responsibility for the document(s) to be reviewed. The author has to ensure that each member of the review group has received a copy of the document in plenty of time prior to the meeting taking place. The purpose of the review is not to criticise the author or his work but is to any find mistakes and to investigate the logic of the code. Unfortunately, authors sometimes take the review personally and do see it as personal criticism.
- **Reviewers** (also called consultants or inspectors). These are technical experts who, after the necessary preparation, take part in the meeting. Their responsibility is to have put in adequate preparation prior to the meeting taking place and to be ready for the review. The reviewer must be

open to new ideas / approaches and they must not be dogmatic. The reviewer has to be aware that the author might take any criticism personally and bear in mind that they are reviewing the document and not the person.

- **Scribe** or minute taker. Documents the deviations, problems, points, suggestions, and so on identified during the meeting. It is important to write clearly and precisely. This person should take as little part in the discussion as possible in order to stay focused on the writing.

Review Management

In order for reviews to be successful it is necessary for them to be correctly managed. This applies particularly to the formal review processes.

- Allow people time to prepare.
- Issue guidance notes (if not in Standards).
- Give plenty of notice.
- Practice Presentations and Walkthroughs.
- Sound out "contentious issues".
- ALWAYS document outcomes and Actions.

When gathering a collection of people together, especially from different departments, it is important that they are properly organised and managed. The person responsible needs to ensure:

- The right people are invited - in good time for them to fit it into their schedules.
- Time for reading and preparation is allowed beforehand.
- Roles are assigned and agreed by all active participants.
- Practice or rehearsals take place to verify timings and to get the messages right.
- For Inspections and Technical Reviews, it is important to arrange that people involved have the right background and expertise. It is essential that they have been trained in the process in which they are about to participate.

3.2.3. Types of Review

There are a number of different types of reviews that can be included as part of the software development life cycle.

Informal Review

A fresh view is always welcome before you embarrass yourself before a wider audience. Asking a colleague / friend to read a document will help with the following:

- Identifying jargon to eliminate or translate.
- Make sure it all makes sense!
- Rationalising the sequence.
- Identifying where the author's familiarity with the subject assumes the reader knows more than (s)he actually does.

The reviewer should be asked to read it without beforehand asking what it is all about - even the most technical document requires an overview, introduction & background. Program code should have comments adequate to make its functionality understandable.

Obviously, the peer should have the requisite technical or business expertise. In other words your postman is probably not the best person to review a network design document - *although he should be good at getting from start to finish in the most economical way*.

Key characteristics:

- no formal process;
- there may be pair programming or a technical lead reviewing designs and code;
- optionally may be documented;
- may vary in usefulness depending on the reviewer;
- main purpose: inexpensive way to get some benefit.

Walkthrough

For some technical deliverables or complex processes it is very useful for the author to walk through the document asking the reviewers to comment and query the concepts and processes that are to be delivered.

The Walkthrough is usually led by the author, although the Project Manager or Team Leader may conduct them. The purpose of a walkthrough is to air ones ideas or approach to a wider audience. It is not the best way to review detail. A Walkthrough is often the precursor to a brainstorming or think tank session.

Often termed Structured Walkthrough; as with any review the participants need to be prepared before the walkthrough and all must know the desired objectives and deliverables of the walkthrough as well as the purpose of the deliverable.

When gathering a collection of people together, especially from different departments, it is important that they are properly organised and managed. The person responsible needs to ensure:

- The right people are invited - in good time for them to fit it into their schedules.
- Time for reading and preparation is allowed beforehand.
- Roles are assigned and agreed by all active participants.
- Practice or rehearsals take place to verify timings and to get the messages right.

Key characteristics:

- meeting led by author;
- scenarios, dry runs, peer group;
- open-ended sessions;
- optionally a pre-meeting preparation of reviewers, review report, list of findings and scribe (who is not the author)
- may vary in practice from quite informal to very formal;
- main purposes: learning, gaining understanding, defect finding.

Technical Review

Although the "ask a friend" technique is helpful, the people you work alongside can also provide a strong defect detection process. These technical reviews are normally more structured than the casual "can you have a quick read of this" approach.

- Often used by workgroups as a self-checking mechanism.

- Its format is defined, agreed upon and practiced with documented processes and outputs.
- Often requires technical expertise.

Technical reviews have more tightly defined objectives than either the walkthrough or the informal review. They are all to do with product verification and validation.

They key objectives are to confirm that

- Product Conforms to Specifications
- Adheres to regulations, guidelines, plans

Clearly for these objectives to be met all reviewers must receive the relevant specifications, guidelines, regulations and plans as part of the review preparation documentation. It should not be assumed that all reviewers will have a complete, or indeed up to date, copy of all relevant documents.

The second set of objectives that technical reviews set out to address is the need to ensure that changes made to existing product are correct and contain no side effects.

- Changes are properly implemented
- Changes affect only those system areas identified by the change specification

Whenever change is introduced to an issued product there is a risk that not only will the change have knock-on effects to other parts of the system or design but also that other areas of the system are changed outside the scope of the specified change. The technical review team must satisfy itself that no additional changes or side effects have been introduced to changed product as well as ensuring that the change has been correctly executed.

If properly defined, it becomes standard practice for certain types of project deliverable to be reviewed by other members of the workgroup - often at a detailed technical level - to identify and eliminate defects.

As with all types of review, the peer review process needs to be properly defined with the organisational, preparation, review and post-review activities all fully documented - ideally with templates and checklists to aid the reviewer and to document the outcomes.

Key characteristics:

- documented, defined defect-detection process that includes peers and technical experts;
- may be performed as a peer review without management participation;
- ideally led by trained moderator (not the author);
- pre-meeting preparation;
- optionally the use of checklists, review report, list of findings and management participation;Preparation of a review report which includes the list of findings, the verdict whether the software product meets its requirements and, where appropriate, recommendations related to findings
- may vary in practice from quite informal to very formal;
- main purposes: discuss, make decisions, evaluate alternatives, find defects, solve technical problems and check conformance to specifications and standards.

Inspections

Michael Fagan worked for IBM in the early 1970's. He devised a method to review program design and code. These inspections are formal events, with a predefined (and unalterable) structure. The code is reviewed by the inspection group following a procedure that engages the developers and others in a formal process of investigation that usually detects more defects in the product - at a lower cost - than does machine testing. Fagan was able to prove the effectiveness of his method and its use is promulgated world-wide. The key features of formal inspections are:

- they are very effective - to the point where they have replaced unit testing.
- they are very formal - Training is required before taking part.
- reviewers must be prepared prior to attendance.
- led by trained moderator (not the author);
- usually peer examination;
- defined roles;
- includes metrics;
- formal process based on rules and checklists with entry and exit criteria;
- pre-meeting preparation;
- inspection report, list of findings;
- formal follow-up process;
- optionally, process improvement and reader;
- main purpose: find defects.

Inspections are a highly structured means to examine, in detail, a deliverable to ensure that it complies with:

- Rules of completeness,
- Standards,
- Preceding document or specification,
- Terms of reference,
- Good Practice,
- Design procedures,
- and so on...

One of the reasons that inspections are so successful is that every participant has a distinct role, when defects are found they are reported (there is no debate) and any contentious issues and queries are resolved outside the inspection and the result reported back to the Moderator. Because of these issues it is necessary for people to be trained (typically 2-3 days) before they can participate in the inspection process.

Each inspection lasts 2 hours max. Some deliverables require several sessions to be fully inspected.

All reviewers will have completed a form that contains all their comments about the deliverable being reviewed. During the Inspection, when they reach the place where a comment is pertinent it is raised. If all agree that it is a defect then it is noted by the moderator.

The author will take a copy of the defects and take the necessary corrective action.

Where the defect is disputed by anyone, the moderator intervenes to ask that it be further investigated outside the Inspection. If it is confirmed as a defect it is remedied. Either way the moderator is informed and takes note of the outcome.

It is noted that people whose work is subject to Inspection take greater care to prepare - thus greatly reducing the number of minor defects - especially those arising from carelessness.

3.2.4. Success factors for reviews

Success factors for reviews include:

- Each review has a clear predefined objective.

  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.eople 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 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.

The most common causes that are responsible for the lack of success in reviews are:

- Roles and responsibilities are not understood or are not taken seriously
- Insufficient training in the review procedures
- Insufficient resources (time, budget, personnel) to carry out reviews
- No general improvement in software development process
- Resistance to formal methods
- Standards and processes to support reviews are either do not exist or are not available in sufficient quality

3.3. Static analysis by tools

What is Static Analysis?

> *"Analysis of software artifacts, e.g. requirements or code, carried out without execution of these software artifacts."*

Static Analysis involves analysing code to identify syntax errors and to generate metrics about the code. It does not involve dynamic execution of the code (hence the "static"). It also assumes that the code meets the business or functional requirements as the Static Analyser cannot check this.

Due to the nature of code (and the size of modern applications) Static Analysis is an automated process - a tool is capable of analysing significant quantities of code much faster than a human.

As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g. control flow and data flow), as well as generated output such as HTML and XML.

Why do it?

Static Analysis is designed to analyse code to look for syntax errors, violations of standards and other possible errors in the code. Some can also analyse the complexity of the program – i.e. its measure of maintainability – and can recommend actions to reduce the complexity level.

Benefits of static analysis

- 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.

Typical defects discovered by static analysis tools include:

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

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 to allow the most effective use of the tool.

Compilers

There are two types of program languages – interpreted and non-interpreted or compiled. Some languages started off as being interpreted, such as Basic, and then compilers were added later to improve performance. Today some languages are debugged while being interpreted and then the tested code is compiled.

Compilers are used to convert the code written by developers into binary that can then be executed by the computer.

Prior to the code being compiled, the compiler will normally perform a syntax check upon the code - such as missing brackets ")" etc. Additionally compilers provide information about the code - i.e. details about variables and memory usage.

Static Analysers

Static Analysers have evolved from compilers - they provide all the same information as compilers plus they perform additional checks on the code and more detailed metrics about it. *See also* Tools for test and test object analysis in section Test Tools.

The one thing that Static Analysers do not do is compilation of the code.

Data Flow Analysis

This is the analysis of the flow of data (i.e. variables) through the application. Variables are storage areas in memory that are given specific names so that they can be easily identified by the developers.

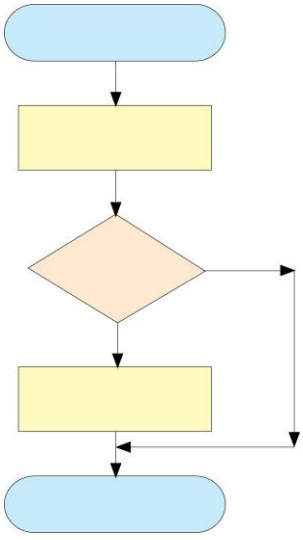
By analysing the data flow the static analyser is able to perform a series of checks on the use of these variables. These checks will highlight such problems with the code as variables being used without having first been declared and variables being accessed after they have been "killed".

Control Flow Analysis

Control flow analysis looks at the way the control moves through the code as the program is executed. This can be used to identify infinite loops in the code, unreachable code and other suspicious aspects in the code.

Control flow analysis produces a control flow graph that graphically shows the flow of control through the component.

Sample Control Flow Graph



Metrics

Static Analysers provide a series of metrics about the code. One of the best known metrics is Cyclomatic Complexity - other metrics produced include Lines of Code (number of lines of code in the component) and Nesting Levels (the number of IF statements that are nested within other IF statements).

Cyclomatic Complexity

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

- L = the number of edges/links in a graph

- N = the number of nodes in a graph

- P = the number of disconnected parts of the graph (e.g. a called graph and a subroutine)

The higher the number the more complex the code and the greater the risk of mistakes and the harder it is to maintain.

3.4. Summary

Static testing

How is static testing done?

By reviewing the documentation & product deliverables.

Why review?

To save resource by identifying errors as soon as possible.

When to review?

As soon as an object is ready, before it is used as a product or the basis for the next step in development.

What to review?

Any deliverable that forms part of the software development life cycle.

Reviews and the Test Process

Reviews are an economical and effective way of finding errors early in the SDLC. Finding an error early means that development costs can be reduced and the system is more likely to go live without significant issues. It is estimated that an on-going review process costs approximately 15% of the development budget.

Features of reviews:

Goals

Validation and verification against specifications and standards, (and process improvement). Achieve consensus.

Activities

Planning, overview meeting, preparation, review meeting, and follow-up (or similar).

Roles and responsibilities

Moderators, authors, reviewers/inspectors and managers (planning activities).

Deliverables

Product changes, source document changes, and improvements (both review and development).

Possible Pitfalls

Insufficient training, insufficient management support (and failure to improve process).

Types of review

Walkthrough

Scenarios, dry runs, used to explain an idea or process, informal, led by the author.

Informal review

Undocumented, informal, but useful, cheap, widely-used, hard to measure benefits of.

Technical (peer) review

Documented, defined fault-detection process, includes peers and technical experts, no management participation. Have a structure therefore are formal reviews.

Inspections

These are formal review processes led by a trained moderator (not the author), use defined roles, include metrics, formal based on rules and checklists with entry and exit criteria.

They are often performed on done on the code itself (through they are not limited to being performed on the code) and in some environments have replaced White Box Testing. Best known – Fagan inspection.

Static Analysis

Static Analysis is the inspection of program logic for mistakes and / or errors i.e. it is a review of the code, without execution. This can help reduce the number of faults in code by examining the syntax of the code and highlighting possible faults. Normally performed by a tool, so is a form of automation.

By reviewing the code's logic, problems of the following nature can be detected:

- Unreachable Code
- Undeclared Variables
- Uncalled Functions or Procedures
- Parameter type mismatches
- Possible array boundary violations

The Static Analysis tool may also produce information on control flow and data flow, these are both concerned with the paths (for control and data items) through the code being reviewed.

Static Analysers also produce metrics on the code being reviewed. Two of the most common metrics are number of lines and cyclomatic complexity (a measure of the complexity of the code).

References

3.2. – IEEE 1028

3.2.2. – Gilb, 1993, van Veenendaal, 2004

3.2.4. – Gilb, 1993, IEEE 1028

3.3. – Van Veenendaal, 2004

4. Test design techniques

4.1. The test development process

The process of identifying test conditions and designing tests consists of a number of recommended steps:

- Analyzing basis test documentation (to identify test conditions);
- Establishing traceability from test conditions back to the specifications;
- Specifying test cases
- Specifying test procedures
- Test implementation

test case specification: A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item.

test design specification: A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases.

test procedure specification: A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.

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

During test design, the test basis documentation is analyzed in order to determine what to test, i.e. to identify the test conditions. A test condition is defined as an item or event that could be verified by one or more test cases (e.g. a function, transaction, quality characteristic or structural element).

Establishing traceability from test conditions back to the specifications and requirements enables both impact analysis, when requirements change, and requirements coverage to be determined for a set of tests. During test design the detailed test approach is implemented based on, among other considerations, the risks identified.

During test case specification the test cases and test data are developed and described in detail by using test design techniques. A test case consists of a set of input values, execution preconditions, expected results and execution post-conditions, developed to cover certain test condition(s). The 'Standard for Software Test Documentation' (IEEE 829) describes the content of test design specifications and test case specifications.

Expected results should be produced as part of the specification of a test case and include outputs, changes to data and states, and any other consequences of the test. If expected results have not been defined then a plausible, but erroneous, result may be interpreted as the correct one. Expected results should ideally be defined prior to test execution.

The test cases are put in an executable order; this is the test procedure specification. The test procedure (or manual test script) specifies the sequence of action for the execution of a test. If tests are run using a test execution tool, the sequence of actions is specified in a test script (which is an automated test procedure).

The various test procedures and automated test scripts are subsequently formed into a test execution schedule that defines the order in which the various test procedures, and possibly automated test scripts, are executed, when they are to be carried out and by whom. The test execution schedule will take into account such factors as regression tests, prioritization, and technical and logical dependencies.

4.2. Categories of test design techniques

The purpose of a test design technique is to identify test conditions and test cases.

Black and White Box Testing

Black box testing techniques (also called specification-based techniques) are appropriate for all phases of testing during the SDLC (i.e. Component Testing through to User Acceptance Testing). Although individual components form part of the structure of a computer system, when you are performing Component Testing it is possible to view the component itself as a black box i.e. design test cases based on its functionality without regard for its structure.

Similarly, white box testing techniques (also called structural or structure-based techniques) can be used at all stages of testing but are used predominately at the Component and Integration Testing in the Small testing phases.

Black box testing and white box testing are terms used to define two different levels of testing involving execution of the code. These require knowledge of the code being tested. Testing without executing the code is known as Static Testing and is covered in a latter sessions.

What is Black Box Testing?

> *"Testing, either functional or non-functional, without reference to the internal structure of the component or system."*

All of us have done this.

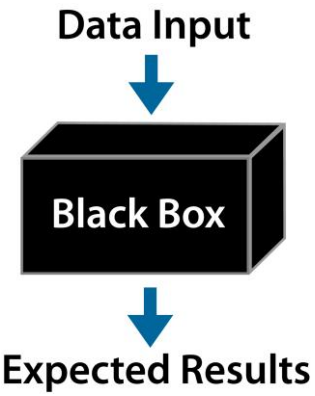
> Q: When we change a light bulb, why do we switch the light on and off again?
>
> A: To make sure it works.
>
> How much do we know about electricity and the wiring involved etc. Similarly what do we all do when our car comes back from a service?

Black box testing concentrates on the functionality of the system - it is not concerned with what goes on underneath the "skin" of the system. As such no knowledge of the code is required.

Black box testing is applicable to all stages of testing, however it is normally concentrated on the latter stages - once component testing is complete.

The various black box testing techniques and their accompanying measurement techniques are detailed in the Software Component Testing Standard BS7925-2. These are covered in the section "Black Box Test Techniques".

Black Box Testing



If Actual Result = Expected Result = Expected Behaviour of SUT then ☑.

What is White Box Testing?

> *"Testing based on an analysis of the internal structure of the component or system."*

White box testing techniques are normally used after an initial set of tests has been derived using black box test techniques. White box testing techniques are usually utilised when measuring the "code coverage" i.e. how much of the program structure has been exercised or covered by a particular set of tests.

Code coverage measurement is best achieved using one of the various tools that are commercially available for the purpose. These code coverage measurement tools can help to increase the productivity and quality. They increase the quality of the code by ensuring that more of the structural aspects are tested, so faults on those structural paths can then be found. They help to increase productivity and efficiency by highlighting any of the test scripts that may be redundant, i.e. testing the same structure as other tests.

White box testing is appropriate for component testing but becomes less useful as testing moves towards system and acceptance testing.

As with all forms of testing white box testing needs to be planned to be effective. There is a test process for component testing - *see* "Component Testing".

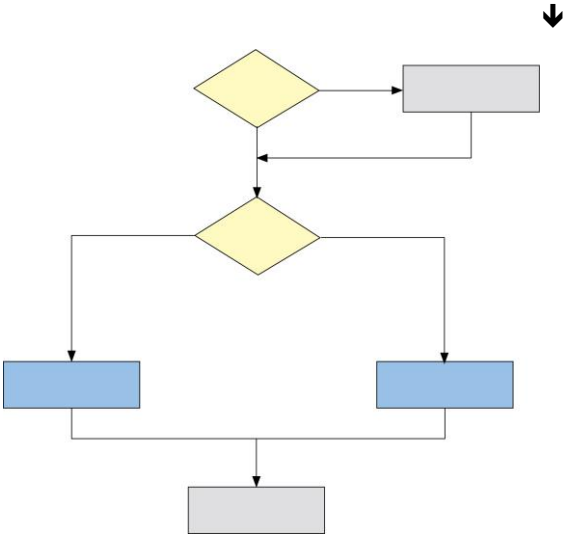
White box testing is capable of thoroughly testing individual components to ensure that they adhere to their specification. It is not aimed at testing the functionality of a component or the interaction of a series of components therefore white box testing has limited use in system and acceptance testing.

However, as developers are often not trained to test (and may be reluctant to do so), component testing is often therefore inadequate. Later stages of testing need to devise and perform tests to address these shortcomings - usually without the skills to examine code or the structure of the program. System and acceptance testers will tend to concentrate on the functionality, specifications and requirements, rather than on the actual code.

The various white box testing techniques and their accompanying measurement techniques are detailed in the Software Component Testing Standard BS7925-2. These will be covered in the session "White Box Test Techniques".

White Box Testing

Decision or Condition



Why do we Need Testing Techniques?

As we have previously mentioned earlier in this course, exhaustive testing is rarely possible and therefore, we have to select a subset of all possible tests. When we select the subset of tests, this subset is only a very tiny subset but it must have a high probability of finding most of the faults in the SUT.

Past experience has shown that selecting a subset at random is neither very effective nor very efficient (even if it is tool supported). When we select subsets of tests, we have to do so by using some intelligent thought process and one such process is that of 'testing techniques'.

What is a Testing Technique?

A testing technique is a thought process that helps us to select a good set of test cases from the total number of all possible tests for a given system. Different techniques provide us with different ways of looking at the SUT and they can even challenge assumptions / widely held beliefs made about the SUT. Each of the techniques provides guidelines for the tester to follow in identifying test conditions and test cases. These guidelines are based on either a behavioural or a structural model of the SUT. In other words, they are based on an understanding of the system's behaviour or its structure, not how it does it.

There are many different testing techniques and those that have been published have been found to be very successful at identifying tests that find faults. The use of testing techniques should be part of the tester's 'best practice' although, it must be remembered that they should not be used to the exclusion of any other approaches.

In layman's terms, a testing technique is a means of identifying good test cases. A good test case consists of four things:

- Effective i.e. has potential to find faults;
- Representative i.e. represents other test cases;
- Modifiable i.e. easy to maintain;
- Cost-effective i.e. doesn't cost much to use.

Advantages of Techniques

As all testers have a different background and go about their testing in a slightly different manner, even when using the same testing technique on the same system, they will invariably arrive at different test cases but they will have a similar probability of finding faults in the SUT. Given the fact that the technique will guide them into having a similar or the same view of the SUT, it makes sense that they will all make similar or the same assumptions when designing test cases for the SUT.

Using various testing techniques allows the testing effort to be more effective and this in turn leads to more faults being found with less effort. Each testing technique will focuses on a particular type of fault and because of this fact, it becomes more likely that the tests that are written will find more of that particular type of fault when they are run. By selecting the appropriate testing techniques, it is possible to control much more accurately what is being tested and thus reduce the likelihood of overlapping between different test cases.

Systematic testing techniques are all measurable and this provides the tester with the ability to quantify the extent of their use, which in turn, makes it possible to gain an objective assessment of the thoroughness of testing with respect to the use of each of the testing techniques. The fact that all the techniques are measurable is useful when comparing one test effort against another and it is helpful when it is necessary to provide the client with confidence in the adequacy of testing that has been undertaken.

Each testing technique falls into one of a number of different categories. Generally speaking, there are only two main categories, which are static and dynamic but, dynamic test techniques are then subdivided into two more categories, which are structural and behavioural. The behavioural test techniques can then be further subdivided into functional and non-functional techniques.

As with all testing activities a systematic approach is vital. Not only does it mean that things are done consistently but it also means that an audit trail exists for the work carried out and that will help improve & increase confidence.

By using these techniques we are developing test cases in a logical manner. With exhaustive testing being impractical this means that we will end up with a sub-set of tests that provide a high probability of finding faults.

Use of testing techniques also enables testers to measure their progress. Because we have a defined, systematic technique with which to create test cases, we should be able to measure the effectiveness of testing and produce metrics to illustrate the state of testing.

What are Coverage Techniques?

Coverage techniques basically serve two purposes - test measurement and test case design. They are often used in the first instance to assess the amount of testing performed by the test cases that have been derived from the functional techniques. Coverage techniques are then used to design additional test cases with the aim of increasing the level of test coverage.

Coverage techniques provide an ideal way of generating additional test cases that are different from the existing test cases. They are also helpful to gauge the breadth of the testing that is being undertaken, in as much as test cases that achieve 100% coverage in any measure will be exercising all parts of the software.

There is also danger in these techniques. 100% coverage does not mean or guarantee 100% tested. Coverage techniques can only measure one dimension of a multi-dimensional concept. For instance, two different test cases may achieve

exactly the same level of coverage but the input data of one test case might find an error whilst the input data of the other test case might not.

In addition, the coverage techniques only measure the coverage of the software code that has been written and it is obvious that they cannot provide any details about the software that has not yet been written. If a function has not yet been implemented in the SUT, only functional testing techniques will reveal the fact.

In common with all the structural testing techniques, coverage techniques are best utilised for areas of software code where more thorough testing is still required. Safety critical code (e.g. the National Air Traffic control system), code that is vital for the correct operation of a system and complex pieces of code are all examples of where structural techniques are particularly useful when testing is being undertaken. They should ***always*** be used in addition to the functional testing techniques rather than as an alternative to them.

The measurement of test coverage can be based on a number of different structural elements in software. The simplest of these measurements is statement coverage, which measures the number of executable statements executed by a set of tests. This is usually expressed in terms of the percentage of all executable statements in the SUT. Statement coverage is the simplest and perhaps the most ineffectual of all coverage techniques, in as much as it is the least likely technique to find errors in the SUT.

Types of Coverage

There are a lot of structural elements that can be used for coverage testing. Each technique uses a different element. Besides statement coverage, there are number of different types of control flow coverage techniques most of which are supported by various tools. These include branch or decision coverage, LCSAJ (linear code sequence and jump) coverage, condition coverage and condition combination coverage.

Another popular, but usually misunderstood coverage measurement is that of path coverage. Path coverage is usually assumed to mean branch or decision coverage because both of these techniques seek to cover 100% of the 'paths' through the code. However, path coverage is technically impossible for any code that contains a loop, since a path that travels round the loop say 5 times is different from the path that travels round the same loop 4 times and this holds true, even if the rest of the paths are identical. Therefore, if it is possible to travel round the loop an infinite number of times, there are also an infinite number of paths through that piece of code. As a result, it is more correct to talk about 'independent path segment coverage' though the shorter term 'path coverage' is frequently used.

Tools

Both black and white box testing benefit from the use of software testing tools. Correctly used tools will increase productivity and the quality of testing should improve. Tools are particularly useful in white box testing where we are examining the code itself.

Common features of black-box (specification-based) techniques:

- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components.
- From these models test cases can be derived systematically.

Common features of structure-based techniques:

- Information about how the software is constructed is used to derive the test cases, for example, code and design.

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

Common features of experience-based techniques:

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

4.3. Specification-based or black box techniques

What is Black Box Testing?

To use a car analogy, black box testing is the equivalent of driving a car. You know how to drive it, but you don't know (or need to know) exactly what happens underneath the bonnet.

In simple terms it means that Test Cases can be designed based on knowledge of **what** the component does without requiring any knowledge as to **how** it does it.

Black box testing is focused on testing the function of the component.

Why do we Need Black Box Test Techniques?

As testers, we will normally expect to be working within tight timescales in which to plan, prepare and exercise our tests. Thus we need to be efficient and optimise our tests wherever possible.

Using a simple edit field as an example:

- The edit field is required to accept a number between (+)1 and (+)100.
- There are one hundred valid tests that should be run - using values 1 through 100.
- There are an infinite number of invalid numeric tests that should be run i.e.101 to infinity, and 0 to minus infinity. (Realistically, there is likely to be a physical limit on the field i.e. it will accept only 3 digits therefore are ranges are 101 to 999 and 0 to -99, giving us 998 invalid tests.)
- There are a further series of invalid tests that should be run - using alphabetic and non-alphanumeric characters.

Thus a simple edit field designed to accept a number between +1 and +100 has a potential of at least hundreds of tests that need to be run against it. In practice we are unlikely to have the time and resources to create and run every one of those tests (not to mention re-running them all as part of a regression pack).

Therefore we need to adapt a series of approaches - test techniques - that enable us to create and use a manageable set of tests that provide coverage of all areas of the component and ensure that some focus is placed upon those areas of the component where errors might occur. This will give us a manageable set of tests that can be created, run and re-run within the limits of testing (i.e. the time and resources available) that still provide maximum test coverage.

4.3.1. Equivalence Partitioning

> *"A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once."*

Equivalence partitioning is based on the premise that the inputs and outputs of a component can be **partitioned into classes** that, according to the component's specification, **will be treated similarly by the component**.

Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.

Initially the equivalence partitions are identified and then test cases derived to exercise the partitions.

Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

Design

Test cases shall be designed to exercise partitions. A test case may exercise any number of partitions. A test case shall comprise the following:

- the input(s) to the component;
- the partitions exercised;
- the expected outcome of the test case.

Test cases are designed to exercise partitions of valid values, and invalid input values. Test cases may also be designed to test that invalid output values cannot be induced.

4.3.2. Boundary Value Analysis

Boundary value analysis is based on the premise that:

a) the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component; and

b) developers are prone to making errors in their treatment of the boundaries of these classes. Thus test cases are generated to exercise these boundaries.

The maximum and minimum values of a partition are its boundary values. A boundary value for a valid partition is a valid boundary value; the boundary of an invalid partition is an invalid boundary value. Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a value on each boundary is chosen.

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect-finding capability is high; detailed specifications are helpful.

Initially the equivalence partitions are identified, then the boundaries of these partitions are identified, and then test cases are derived to exercise the boundaries.

Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

The general approach with Boundary Value Analysis is to select values **just on** the boundary, **on** the boundary and **just below** the boundary.

Design

Test cases shall be designed to exercise values both on and next to the boundaries of the partitions. For each identified boundary three test cases shall be produced corresponding to values on the boundary and an incremental distance either side of it. This incremental distance is defined as the smallest significant value for the data type under consideration.

A test case shall comprise the following:

- the input(s) to the component;
- the partition boundaries exercised;
- the expected outcome of the test case.

Test cases are designed to exercise valid boundary values, and invalid input boundary values. Test cases may also be designed to test that invalid output boundary values cannot be induced.

Why do both Equivalence Partitioning and Boundary Value Analysis?

Technically speaking, if you only performed boundary value analysis (BVA) you would also have tested every equivalence partition (EP) because every boundary is in a partition as well. However, this approach will cause problems when the value fails - was it only the boundary value that failed or did the whole partition fail?

In addition, by testing only the boundaries, we would probably not provide the users with that much confidence in the testing, as we are only using extreme values rather than normal values.

As a consequence, it is recommended that the partitions are tested separately from the boundaries - this requires the tester chooses partition values that are NOT boundary values.

Which partitions and boundaries you decide to exercise and the order, in which you do so, really depends on your objectives. If your ultimate goal is to utilise the most thorough approach, then it is recommended that you follow the traditional approach.

The traditional approach states that you test the valid partitions first, followed by the invalid partitions, then the valid boundaries and finally the invalid boundaries. If however, you are under time pressure and cannot test everything (which is the norm for most projects), then your objective will help you decide what to test.

For instance, if you are after user confidence by running the minimum number of tests, you may decide to only perform valid partition tests. However, if you want to find as many faults as possible in the least possible time, you may start with testing the invalid boundaries.

Exercises

Perform Equivalence Partitioning and Boundary Value Analysis for the following:

Question 1

To be eligible for a mortgage you must be between the ages of 18 and 64 (inclusive). The age input field will only accept two digits and will not accept minus figures ("-"). What are the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis?

Question 2

An input field on a mortgage calculator requires a value between 15,000 and 2,000,000. The field only allows numerical values to be entered and has a

maximum length of 9 digits. What are the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis?

Question 3

The term of a mortgage can be between 5 and 30 years, identify the valid values for Equivalence Partitioning and Boundary Value Analysis?

Question 4

The font formatting box in a word processing package allows the user to select the size of the font – ranging from 6 point to 72 point (in 0.5 steps).

What are the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis?

Question 5

A screen for entering mortgage applications requires information on both peoples wages and will generate the maximum amount available for the mortgage (based on 3¼ times larger wage, 1¼ times lower wage). If the mortgage is less than £250,000 then the interest rate is 4.5%, if the amount is £250,000 to £1,000,000 then the interest rate is 4%.

What are the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis necessary to test the output (i.e. the interest rate)?

Question 6

Personal loan of between £1000 and £25000. For loans between £1,000 and £10,000 there is an interest rate of 8.5%, loans between £10,001 and £25,000 have an interest rate of 8%.

What are the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis?

Question 7

A grading system takes student marks (coursework 0 – 75 and exam 0 – 25) and generates a grade based on those marks (0 – 40 Fail, 41 – 60 C, 61 – 80 B and 81 – 100 A).

Identify the valid and invalid values for Equivalence Partitioning and Boundary Value Analysis necessary to test the output (i.e. the grade).

4.3.3. Decision table testing

The method uses a model of the relationships between causes and effects for the component. Each cause is expressed as a Boolean (true or false) condition on an input, or combination of inputs. Each effect is expressed as a Boolean expression representing an outcome, or a combination of outcomes.

The model is represented as a Boolean graph relating the derived input and output expressions using the operators: AND, OR, NAND, NOR, NOT. From this graph, or even without ever getting to the graph, a decision table is produced. The table represents the logical relationships between causes and effects.

Test cases are designed to exercise rules, which define the relationship between the component's inputs and outputs, where each rule corresponds to a unique possible combination of inputs to the component. For each test case the true or false state for each cause and each effect is identified.

Decision tables are good for:

- capturing system requirements that contain logical conditions;

- documenting internal system design;
- recording complex business rules that a system is to implement;
- serving as a guide to creating test cases that otherwise might not be exercised.

Each row of the decision table is a condition (cause) or action (effect). Each column of the table is a business rule that defines a unique combination of conditions that result in the execution of the actions associated with that rule. When designing test cases we aim at having at least one test per column, which typically involves covering all combinations of triggering conditions.

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

Example

(With modifications from the BS7925-2.)

A check debit function has **inputs**:

- debit amount,
- account type

  p = postal

  c = counter
- current balance

And **outputs** are:

- new balance and
- action code =

  D&L = process debit and send out letter

  D = process debit only

  S&L = suspend account and send out letter

  L = send out letter only

The function has the following specification:

If there are sufficient funds available in the account or the new balance would be within the authorized overdraft limit then the debit is processed. If the new balance would exceed the authorized overdraft limit then the debit is not processed and if it is a postal account it is suspended. Letters are sent out for all transactions on postal accounts and for non-postal accounts if there are insufficient funds available (i.e. the account would no longer be in credit).

The conditions are:

C1 New balance in credit

C2 New balance overdraft, but within authorized limit

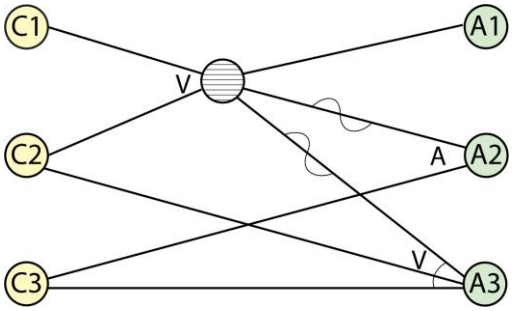
C3 Account is postal

The actions are:

A1 Process debit

A2 Suspend account

A3 Send out letter



The cause-effect graph shows the relationship between the conditions.

The code graph is then recast in terms of a decision table. Each column of the decision table is a rule. The table comprises two parts. In the first part each rule is tabulated against the conditions. A 'T' indicates that the condition must be TRUE for the rule to apply and an 'F' indicates that the condition must be FALSE for the rule to apply. In the second part, each rule is tabulated against the actions. A 'T' indicates that the action will be performed; an 'F' indicates that the action will not be performed; an asterisk (*) indicates that the combination of conditions is infeasible, so no actions are defined.

The example has the following decision table:

Rules	1	2	3	4	5	6	7	8
C1: New balance in credit	F	F	F	F	T	T	T	T
C2: New balance overdraft, but within authorised limit	F	F	T	T	F	F	T	T
C3: Account is postal	F	T	F	T	T	T	F	T
A1: Process debit	F	F	T	T	T	T	*	*
A2: Suspend account	F	T	F	F	F	F	*	*
A3: Send out letter	T	T	T	T	T	T	*	*

The following test cases are required to provide 100% cause-effect coverage, and correspond to the rules in the decision table above. Test cases are not generated for rules 7 and 8 as they are infeasible.

Test Case	Causes				Effects	
	Account Type	Overdraft Limit	Current Balance	Debit Amount	New Balance	Action Code
1	'c'	£100	-£70	£50	-£70	'L'
2	'p'	£1500	£420	£2000	£420	'S&L'
3	'c'	£250	£650	£800	£150	'D&L'
4	'p'	£750	£500	£200	£700	'D&L'
5	'c'	£1000	£2100	£1200	£900	'D'
6	'p'	£500	£250	£150	£100	'D&L'

4.3.4. State Transition

A system may exhibit a different response depending on current conditions or previous history (its state). In this case, that aspect of the system can be shown as a state transition diagram. It allows the tester to view the software in terms of its states, transitions between states, the inputs or events that trigger state changes (transitions) and the actions which may result from those transitions.

This is a function based technique that uses an analysis of the specification of a component to model its behaviour by state transitions.

It looks at:

- The various states that a component may occupy.
- The transition between those states.
- The events that cause those transitions.
- The actions that may result from those transitions.

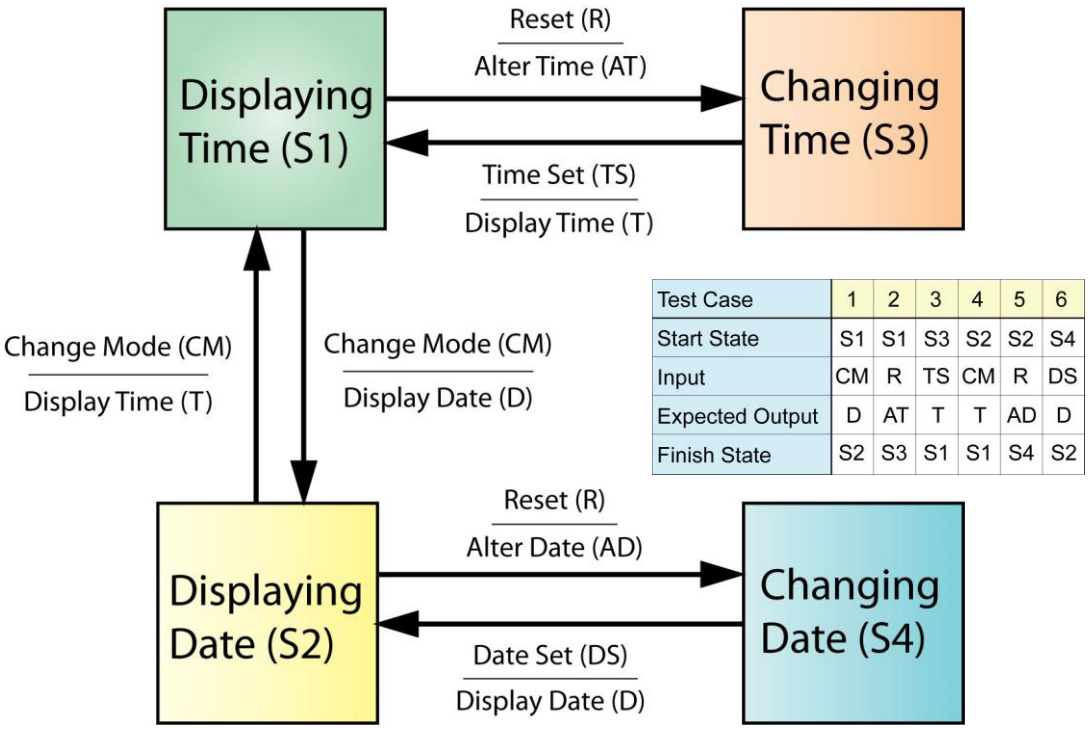
The technique requires that we identify (from the components specification):

- The various states the component may occupy.
- The possible combinations of states (including from one state to another and immediately back again).
- The events that cause the transitions.
- The inputs to the component that will trigger the event.
- Any outputs from the component caused by the transition.

State Transitions will typically be shown as a state transition diagram, a state transition model or a state table.

State transition testing is much used within the embedded software industry and 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).

Example:



Test Case	1	2	3	4	5	6
Start State	S1	S1	S3	S2	S2	S4
Input	CM	R	TS	CM	R	DS
Expected Output	D	AT	T	T	AD	D
Finish State	S2	S3	S1	S1	S4	S2

4.3.5. Use case testing

The syllabus lists Use case testing as a Black-box technique. It is mentioned in the handouts in section 4.2.3 Business-Process-Based Testing. Another term is Transaction testing.

This is an entire approach to testing rather than a technique. The point is that tests are designed based on business activities as opposed to based on requirements.

At least one test case is created to cover the main scenario of the use cases. In addition at least one test case is created for each extension, if any.

Tests can be specified from use cases or business scenarios. A use case describes interactions between actors, including users and the system, which produce a result of value to a system user. Each use case has preconditions, which need to be met for a use case to work successfully. Each use case terminates with post-conditions, which are the observable results and final state of the system after the use case has been completed. A use case usually has a mainstream (i.e. most likely) scenario, and sometimes alternative branches.

Use cases describe the "process flows" through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system. Use cases, often referred to as scenarios, are very useful for designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not see.

Other black-box techniques

- **Syntax Testing -** Uses the syntax of the components' inputs as the basis for the design of the test case.
- **Random Testing -** This black box technique requires no partitioning of the component's inputs, but simply requires test inputs to be chosen from at random.

4.4. Structure-based or white box techniques

What is White Box testing?

> *"Testing based on an analysis of the internal structure of the component or system."*

To use a car analogy, White box testing is getting underneath the bonnet, so we need to know how the vehicle works in order to test it.

White box testing is concerned with testing the code within a component. As such it requires knowledge of the internal workings of a component. Hence, it tends to be used extensively in the early stages of testing becoming less used as the testing draws to a conclusion.

Structure-based testing/white-box testing is based on an identified structure of the software or system, as seen in the following examples:

- Component level: the structure is that of the code itself, i.e. statements, decisions or branches.
- Integration level: the structure may be a call tree (a diagram in which modules call other modules).
- System level: the structure may be a menu structure, business process or web page structure.

Why do we Need White Box Test Techniques?

We use White Box Test Techniques to formally structure testing of a component and to measure how much of a component has been tested.

We need to structure how we test code to ensure maximum coverage of tests - i.e. maximum amount of each is covered, all branches are tested etc. White box testing doesn't care about the functionality of code - i.e. business usage - hence tests cases could be created that cannot actually occur in usage of the application.

There are a range of White Box Test Techniques and each provides a different way of testing the code and each has its own testing measurement technique.

White box testing is best done using tools - to execute a single component we need to create a "wrapper" that will emulate any other components needed, to feed data in and to take data out. Many components do not have a GUI interface therefore there is no obvious way to interact with them.

4.4.1. Statement testing and coverage

Analysis

Statement testing uses a model of the source code which identifies statements as either executable or non-executable.

Design

Test cases shall be designed to exercise executable statements.

For each test case, the following shall be specified:

- the input(s) to the component;
- identification of statement(s) to be executed by the test case;
- the expected outcome of the test case.

This structural test technique is based upon the decomposition of the component into constituent statements. The purpose of statement testing is to build test cases that execute every line of executable code within the component.

The two principal questions to answer are:

- What is a statement?
- Which statements are executable?

In general, a statement should be executed completely, or not at all. For instance:

IF *a* THEN *b* ENDIF

Is considered more than one statement because *b* may or may not be executed depending upon the condition *a*. The definition of *statement* used for statement testing may not be the one used in the language definition.

Consider the following code:

```
a;
if b;
    c;
d;
```

where a; b; c; d; are statements.

Any test case with *b* TRUE will achieve full statement coverage. Note that full statement coverage can be achieved without *b* FALSE.

Example 2:

Consider the following code:

```
X = Prompt ('Press any key')
If X > 5 then
  Print Message 'Greater than 5'
End If
```

The code above prompts the user for an input, which if the input is greater than 5, a message will be printed.

To exercise every line of program code, we need to enter a numeric value greater than 5. Therefore, if we enter a value of 7, every line of code will have been exercised but not every path through it.

Statement coverage has a number of weaknesses in that it only tests one of the possible outcomes – in this instance where X > 5, but the outcome for X < 5 has not been covered.

To determine statement coverage (the amount of code that has been executed by the tests) we use the following formula:

Statement coverage =
Number of executable statements executed
*
100% Number of executable statements

See BS7925-2 Software Component Testing standard sections 3.6, 4.6 and B.6 for further details and examples.

4.4.2. Decision testing and coverage

Analysis

Decision testing requires a model of the source code which identifies decisions and decision outcomes. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. Typical decisions are found in loop and selections. Each possible transfer of control is a decision outcome.

Design

Test cases shall be designed to exercise decision outcomes.

For each test case, we shall specify:

- the input(s) to the component;
- identification of decision outcome(s) to be executed by the test case;
- the expected outcome of the test case.

Decision coverage may be demonstrated through coverage of the control flow graph of the program. The first step to constructing a control flow graph for a given procedure is to divide it into basic blocks. These are sequences of instructions with no branches into the block (except to the beginning) and no branches out of the block (except at the end). The statements in a basic block are guaranteed to be executed together or not at all.

Example

If we consider the same code as in Example 2 in Statement Testing:

```
X = Prompt ('Press any key')
If X > 5 then
  Print Message 'Greater than 5'
End If
```

If we now subject the above code to branch coverage, we have to exercise the code for both the TRUE (X > 5) and the FALSE (X < 5) options. Therefore, 2 test cases are necessary to provide test coverage for decision testing i.e. X = 6 and X = 4.

Decision coverage is considered the next logical coverage progression from statement coverage. However, decision coverage is still unable to cover 100% of the code (no structural test method can) and it is not suitable for testing the more complex, compound condition e.g. If X > 5 or (X < Y and Y – X = 3) THEN…

Please Note: 100% decision coverage guarantees 100% statement coverage. Branch and decision coverage are actually slightly different for less than 100% coverage, but at 100% coverage they give exactly the same results.

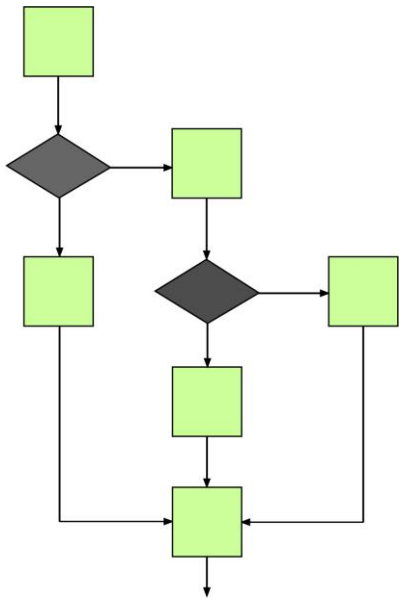
Exercises

Identify the number of tests required to achieve:

1. 100% statement coverage.
2. 100% decision coverage.

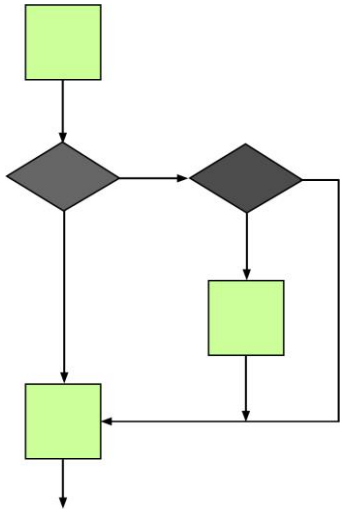
Question 1

```
enter user ID
IF user ID is valid THEN
        display "enter password"
        IF password is valid THEN
                display account screen
        ELSE
                display "wrong password"
ELSE
        display "wrong ID"
END IF
display time & date
```



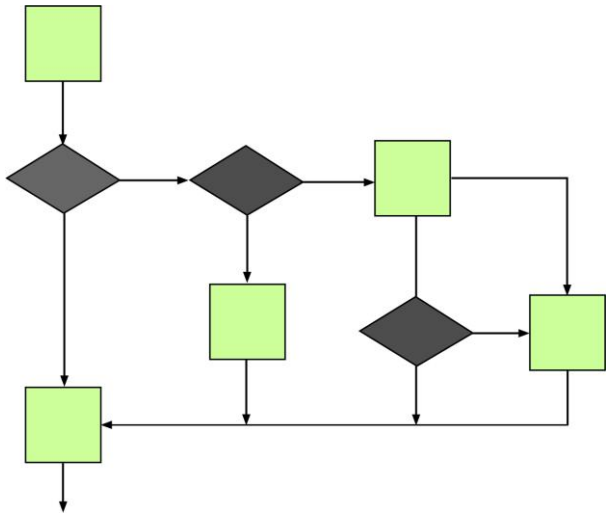
Question 2

```
READ AGE
IF AGE>0 THEN
        IF AGE=21 THEN
                PRINT "21st"
        END IF
END IF
PRINT AGE
```



Question 3

```
READ AGE
READ BIRTHYEAR
IF AGE>0 THEN
        IF BIRTHYEAR =0 THEN
                PRINT "No values"
        ELSE
                PRINT BIRTHYEAR
                IF AGE>21 THEN
                        PRINT AGE
                END IF
        END IF
END IF
READ BIRTHMONTH
```



Question 4

```
READ HUSBANDAGE
READ WIFEAGE
IF HUSBANDAGE>65
        PRINT "Husband retired"
ELSE
        PRINT "Husband not retired"
END IF
IF WIFEAGE>65
        PRINT "Wife retired"
ELSE
        PRINT "Wife not retired"
END IF
```

Question 5

```
READ HUSBANDAGE
READ WIFEAGE
IF HUSBANDAGE>65
        PRINT "Husband retired"
END IF
IF WIFEAGE>65
        PRINT "Wife retired"
END IF
```

Question 6

```
READ HUSBANDAGE
IF HUSBANDAGE<65
        PRINT "Below 65"
END IF
IF HUSBANDAGE>64
        PRINT "Above 64"
END IF
```

4.4.3. Other structure-based techniques

Branch Condition Testing

Branch Condition Testing requires a model of the source code which identifies decisions and the individual Boolean operands within the decision conditions. A decision is an executable statement which may transfer control to another statement depending upon the logic of the decision statement. A decision condition is a Boolean expression which is evaluated to determine the outcome of a decision. Typical decisions are found in loops and selections.

Branch Condition Combination Testing

Branch condition combination coverage would require all combinations of Boolean operands A, B and C to be evaluated.

Branch condition combination coverage is very thorough, requiring 2*n* test cases to achieve 100% coverage of a condition containing *n* Boolean operands. This rapidly becomes unachievable for more complex conditions.

Modified Condition Decision Testing

Modified condition decision coverage (MCDC) is a pragmatic compromise that requires fewer test cases than branch condition combination coverage.

Modified condition decision coverage requires test cases to show that each Boolean operand (A, B and C) can independently affect the outcome of the decision. This is less than all the combinations (as required by branch condition combination coverage).

Linear Code Sequence and Jump

An LCSAJ (Linear Code Sequence & Jump) is a linear sequence of executable code commencing either from the start of a program or from a point to which control flow may jump, and terminated either by a specific control flow jump or by the end of the program. It may contain predicates that have to be satisfied in order to execute the linear code sequence and terminating jump.

Data Flow Testing

Data Flow Testing uses a model of the interactions between parts of a component connected by the flow of data as well as the flow of control.

Categories are assigned to variable occurrences in the component, where the category identifies the definition or the use of the variable at that point. Definitions are variable occurrences where a variable is given a new value, and uses are variable occurrences where a variable is not given a new value, although uses can be further distinguished as either data definition P-uses or data definition C-uses. Data definition P-uses occur in the predicate portion of a decision statement such as while .. do, if .. then .. else, etc. Data definition C-uses are all others, including variable occurrences in the right hand side of an assignment statement, or an output statement.

The control flow model for the component is derived and the location and category of variable occurrences on it are identified.

4.5. Experience-based techniques

4.5.1. Error guessing

> *"A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them."*

Error guessing is referred to, by a number of different names including experience-driven testing, heuristic testing and lateral testing.

Error Guessing is a technique used by testers that enables them to utilise their own experiences to 'predict' where errors may occur.

It should not be used as a primary method for planning and preparing tests but it is more than useful method by which latent errors can be detected.

Testers can and should use their past experience to guess or predict where errors are likely to occur.

It should be used to compliment the more systematic test design techniques and not instead of them.

It should be used as a "mopping up" technique rather than as a first choice technique.

You may have experience in the insurance field that suggests that the number of factors influencing the calculation of Car Insurance no claims bonuses is vast and that using boundary value analysis or other more formal approaches tends to leave gaps in the testing. You may feel you need to run additional tests to check these out.

Similarly the number of factors these days that impact social security benefits is constantly changing, again boundary values may leave a number of areas exposed in your experience and additional tests would always help!

You may have experience of testing a particular developer's code and know that there are some inherent weaknesses that may not be exposed by more structured techniques. For example a developer may not 're-cycle' all the storage used while running his code, leaving potential storage leaks in the application. These may not get detected using normal testing techniques so you run additional checks to ensure that most effective use and replacement of storage is achieved.

You may feel that there are gaps in the development life cycle and decide to run extra tests to try to cover these short-falls. These tests will raise errors - causal analysis will show a shortcoming in the development process, and the process will be fixed (we hope).

Your experience shows that storage management under Windows 95 is lacking. While tests carried out in an NT environment may run OK and highlight no real storage concerns your experience directs you to execute the same tests in a 95 environment to highlight the performance/storage differences.

Although Error Guessing may appear "haphazard", it still needs to be planned, it is not a form of "ad-hoc" testing.

Separate to that, Ad-Hoc testing can be encouraged if:

- there is sufficient time;
- a log of tests is maintained:
- the tester is able to recall what actions were performed during the tests.

Without the ability to re-enact exact what was done, the tester cannot repeat the actions that lead to the fault occurring (assuming it was a fault and not user error). Without this reproducibility or a detailed description of ***exactly*** the actions taken, the cause of the fault is not traceable and therefore the developers are unlikely to be able to fix it.

4.5.2. Exploratory Testing

Exploratory software testing can be a powerful approach to testing. It can sometimes be much more productive than testing using scripts. Most testers perform exploratory testing at one time or another. However, it is not generally well respected in testing circles.

Exploratory testing is test design and test execution at the same time, as opposed to scripted testing (predefined test procedures, whether manual or automated). Exploratory tests are not defined in advance and carried out precisely according to a plan. In practice, this is not a simple distinction. Even well defined test procedure leaves certain details to the discretion of the tester (e.g. how quickly to type at the keyboard). An exploratory test session will involve tacit constraints concerning which parts of the product are to be tested, or what strategies are to be used. Good exploratory testers will write down test ideas and use them in later test cycles. Although these notes may look like test scripts, they are not.

The next test is influenced by the result of the previous one, and to this extent, this is an exploratory testing. This technique becomes more prevalent when you cannot tell in advance of the test cycle what tests should be run, or when there has not been enough time to create those tests. When running scripted tests, and new information comes to light that suggests a better test way of doing things, switching to exploratory testing may be indicated. A more scripted approach is appropriate when there is more certainty about how testing should be conducted, new tests are relatively unimportant, the need for efficiency and reliability in executing those tests is worth the effort of scripting, and when the cost of documenting and maintaining tests is worth paying.

The results of exploratory testing aren't particularly different from those from scripted testing, and the two approaches are compatible.

Some people say that writing down test scripts and following them disrupts the intellectual processes that make testers able quickly to find important problems. The more intellectually challenging and flowing the test process is, the more likely the right tests will be found at the right time. Exploratory testing's power comes from the richness of this process being limited only by the breadth and depth testers' imaginations and insights gained into the nature of the system being tested.

Scripting is still important. Some testing situations need to be repeatable and very efficient, so they should be scripted or automated. This may be the case where a test environment is available intermittently. Exploratory testing is especially useful in complex testing situations, when little is known about the product, or as part of preparing a set of scripted tests. It is called for any time the next test you should perform is not obvious, or when you want to go beyond the obvious.

4.6. Choosing test techniques

The choice of a test technique may be made based on:

- Regulatory standards
- Experience and / or
- Customer / contractual requirements

Selection factors

The following selection factors are important:

- Risk level and risk type (thoroughness)
- Test objective
- Documentation available

- Knowledge of the test engineers
- Time and budget
- Supporting tools – for design/regression testing
- Product life cycle status
- Previous experiences
- Measurements on test effectiveness
- International standards – many times they are industry specific
- Customer/contractual requirements
- Auditability/traceability
- A lot of information, skills necessary!

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

4.7. Summary

Common features of black-box (specification-based) techniques:

- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components.
- From these models test cases can be derived systematically.

Common features of structure-based techniques:

- Information about how the software is constructed is used to derive the test cases, for example, code and design.
- The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage.

Common features of experience-based techniques:

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

Black & White Box Testing

- **Black Box testing**

  "Testing, either functional or non-functional, without reference to the internal structure of the component or system."

- **White Box testing**

  "Testing based on an analysis of the internal structure of the component or system."

Black Box or Function Based testing is about testing the specification of a system or system component and it is not being concerned with precisely 'HOW' it does it but much more concerned with 'WHAT' it does.

White Box testing is about testing the precise construction details of the design. Tests cases are built from a detailed knowledge of the internal workings of the system. In White box testing whilst you are concerned with 'WHAT', the main focus of the testing is 'HOW' a particular piece of code works.

Black and White Box testing have their uses in different parts of the development Life-cycle. White Box testing is most often used in Unit and Link testing, while Black Box testing is of better use in system testing. However, both techniques can be used throughout the SDLC.

Black Box Test Techniques

- **Cause Effect Graphing** – Identifies logical relationships between causes and effects for the component.
- **Equivalence Partition** – Partitions the input and output values. Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered "equivalent").
- **Boundary Value Analysis** – Partitions the input and output values of the component in to a number of ordered sets with identified boundaries. Tests are identified for values immediately above the boundary, on the boundary and immediately below the boundary.
- **State Transition** – Transition of component / system between visible states.
- **Use case testing** - Tests can be specified from use cases or business scenarios. A use case describes interactions between actors, including users and the system, which produce a result of value to a system user. Use cases describe the "process flows" through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system.
- **Syntax testing** – Test the syntax of the inputs to a component.
- **Random Testing** – Create tests with fixed steps but random data (i.e. chosen by the tester as the test is being executed).

All black box techniques have a systematic approach and all, except Syntax and Random, have a corresponding measurement technique.

White Box Test Techniques

- **Statement testing** – Executes all statements within a component.
- **Branch/decision testing** – A technique used to execute all branches the code may take based on decisions made.
- **Branch condition testing** – Condition testing is based upon the analysis of conditional control flow within the component. Three types – Branch Condition testing, Branch Condition Combination testing & Modified Condition Combination testing.
- **Linear code sequence and jump** – An LCSAJ is defined as a linear sequence of executable code. Commencing either from the start of a program or from a point to which control flow may jump. Terminated either by a specific control flow jump or by the end of the program.

- **Data flow testing** – Interaction between parts of a component connected by the flow of data (as well as the flow of control).

All white box techniques have a systematic approach and a corresponding measurement technique.

Experience-based techniques

- **Error guessing**

  Error Guessing is a technique used by testers that enables them to utilise their own experience to 'predict' where errors may occur. Can detect some faults that systematic techniques can miss. Test cases are derived from experience of where errors have occurred in the past or the tester has an insight as to where errors are likely to occur in the future.

  It should not be used as a primary method for planning and preparing tests but is a more than useful method when it comes to detecting latent errors.

  Although Error Guessing may appear "haphazard", it still needs to be planned; it is not a form of "ad-hoc" testing.

- **Exploratory testing**

  Exploratory testing is concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes. It is an approach that is most useful where there are a few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing.

References

4.1. – Craig, 2002, Hetzel, 1988, IEEE 829

4.2. – Beizer, 1990, Copeland, 2004

4.3.1. – Copeland, 2004, Myers, 1979

4.3.2. – Copeland, 2004, Myers, 1979

4.3.3. – Beizer, 1990, Copeland, 2004

4.3.4. – Beizer, 1990, Copeland, 2004

4.3.5. – Copeland, 2004

4.4.3. – Beizer, 1990, Copeland, 2004

4.5. – Kaner, 2002

4.6. – Beizer, 1990, Copeland, 2004

5. Test Management

5.1. Test organization

Every testing project will be unique in some manner. There are a range of standard approaches to the organisation of testing. Each has its own advantages & disadvantages and offer varying degrees of testing independence.

5.1.1. Test organization and test independence

Identify what needs to be tested and by whom

The various phases of testing that a system will pass through on its way to going 'live' need to be identified and the responsibility for this testing needs to be assigned. In an ideal world, the system should pass though all the phases discussed in this course but due to various factors, in the 'real world' this does not always happen / is not always possible. Knowing exactly what testing can be achieved within the allocated time frame is important so that the work load / tasks can be adjusted accordingly.

An important consideration is who will actually undertake the task of testing specific phases / areas throughout the SDLC. A list of people who might be responsible for undertaking testing at various stages in the SDLC follows:

- The developer.
- Independent testers within the development teams.
- Independent test team or group within the organization, reporting to project management or executive management.
- Independent testers from the business organization, user community and IT.
- Independent test specialists for specific test targets such as usability testers, security testers or certification testers (who certify a software product against standards and regulations).
- Independent testers outsourced or external to the organization.

The advantages and disadvantages of each method are detailed below.

Developers themselves do the testing

This is prevalent in organisations where the main driver is the rapid turn around and delivery of software.

Testing as a separate discipline is regarded as an unnecessary delay to delivery. It is done as an afterthought with little planning and the developers are responsible that it is carried out.

Developers will tend to confirm the functionality they have coded rather than look for errors in that functionality.

One of the major risks of a developer testing his / her own work is that any assumptions made during development will continue to be made during testing.

One method of reducing this risk is to have each developer test a colleagues work and to have their own work tested by a colleague. This is known as "buddy testing".

A tester on the development team

Another approach is to assign a dedicated tester to each project team.

This person is responsible for testing – (s)he will not get involved with the actual developing.

This offers the major advantages that the tester can be involved throughout the development life cycle and should be able to easily talk directly to the developers (because they are part of the same team).

The major disadvantage of this approach is that the tester may be too closely involved with the team to provide an objective viewpoint. For example, any assumptions made by the developers may be accepted by the tester because s/he was present when those assumptions were first made by the team.

Independent test team or group within the organization

A group of dedicated testers may be in place, they have responsibility for testing the developed software.

There will be more inclined to look at the negative aspects of the SUT.

The tendency here is that the test team only get involved later on in the project, by which time the pressure is on and there is little scope for radical changes to the application. The resultant implementation could be a compromise.

Independent test specialists

The organisation may have a group of internal 'testing consultants' who offer advice and support to the projects, throughout the life cycle, for example usability testers, security testers or certification testers.

These tend to be small groups whose resources are stretched, again the onus is with the project manager to either take or ignore the advice offered.

Outsourcing testing

Outsourcing either all or significant parts of testing to an independent testing agency will guarantee that objective testing is carried out.

Usual obstacles to this set up are the cost and how can an independent body have empathy with the business sector the client is operating in?

The benefits of independence include:

- Independent testers see other and different defects, and are unbiased.
- An independent tester can verify assumptions people made during specification and implementation of the system.

Drawbacks include:

- Isolation from the development team (if treated as totally independent).
- Independent testers may be the bottleneck as the last checkpoint.
- Developers lose a sense of responsibility for quality.

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, developer, business and domain expert, infrastructure or IT operations.

5.1.2. Tasks of the Test Leader and tester

The multi-disciplined team needs to be able to, if necessary, cover all aspects of testing as noted in these lists. For complete outsourcing of testing, all these skills need to be present. Even if only partial outsourcing is required, an awareness of these skills is needed.

In these days of Web application projects, which are known as Rapid Application Design (RAD) projects, the system will be subjected to frequent and constant changes. To ensure that the project is kept on track, which is vital as speed to market, is important, very experienced and skilled testers are going to be necessary. Without experienced testers, the project deadlines will constantly slip and vital testing might be overlooked.

Besides technically skilled and knowledgeable testers must also be socially competent, good team players, willing and ready to query in depth, assertive, confident, exact and creative, analytical. They also need political and diplomatic skills and be capable of quickly familiarising themselves with complex implementations and applications. *See also* Psychology of Testing in section Basics of Software Testing.

Testing teams will typically include

- Test analysts to prepare, execute and analyse tests
- Test Consultants prepare strategies and plans
- Test automation experts
- Load & performance experts
- Database administrator or designer
- Test managers / team leaders
- Test environment management
- … and others as needed

Tasks of the Test Manager

The test manager has to be the engine and mind of the testing effort. The various tasks and activities include:

- Coordinate the test strategy and plan with project managers and others
- Write or review a test strategy for the project, and test policy for the organization.
- Contribute the testing perspective to other project activities, such as integration planning.
- Estimate the time and cost of testing.
- Determine the needed skills of the team.
- Plan for and acquire the necessary resources.
- Negotiate and set test completion criteria.
- Define which test cases, by which tester, in which sequence and at which point in time are to be carried out.
- Adapt the test plan to the results and progress of the testing.
- Introduce suitable compensatory measures (in case of delays, absences, leaves).
- Decide what should be automated, to what degree, and how.
- Select tools to support testing and organize any training in tool use for testers.
- Decide about the implementation of the test environment.

- Schedule tests.
- Report the current test results and test progress.
- Decide when the tests can be completed based on test completion criteria.
- Introduce and use suitable metrics for measuring and rating test progress and test completion.

Determine the Requisite Skill Sets

The skill sets required will depend on the type of project, the complexity of the code, the frequency of changes and the complexity of the SUT. Sometimes the required skill sets cannot be obtained and in these instances, compromises have to be made. If, for example, the test team is particularly inexperienced or unfamiliar with the SUT, then the test scripts should be more detailed than usual. This will result in a longer period of time being spent in preparing the scripts to be executed, and indeed, the execution of these scripts will take the test team longer.

Provide the required Resources (both Internal and External)

Resources in this instance, includes the people undertaking the testing, the time available to test, securing the budget to fund the testing activities and the creation of the test environment / systems. It must be borne in mind that as the testing enters the different phases in the SDLC, additional / separate test environments / systems might be required. In an ideal world, separate test environments / systems would be available for the various stages of testing, as data conflicts from different test teams might corrupt the test results.

Testing should, as far as practicable reflect the following for each level of testing:

- Acceptance testing - the operational environment.
- Integration testing - the physical operational interfaces of the 'live' system.
- System testing - the operational environment.
- Unit testing - usually divorced from the operational environment.
- Regression testing - the operational environment.

If the people undertaking the testing are employees of the company but actually work for other departments, arrangements need to be made for these staff to be released to the test team for the duration of the test activities. Alternatively if they are external contract staff, then the rates of remuneration and the length of contracts need to be agreed.

Decide on Project Reporting Lines

Reporting is by-directional: the test manager will report the current test results and test progress to the client or upper management. The test manager will also receive reports from the testers.

It is important that people are aware of the reporting lines from the inception so that the primary objective of getting the system delivered on time and within budget is maintained / achieved. The organisation must make both the development team and the testing team aware of the reporting structure for the project, so that there is no ambiguity concerning who reports to whom. Otherwise confusions can occur when both teams believe that they have priority over the other and this can lead to unnecessary complications.

Introduce suitable compensatory measures

When there are schedule delays and personnel absences and leaves a project might go wrong. The test manager needs to take adequate compensatory

measures. It may be necessary to employ additional resources (personnel, work places, tools) in order to make up the lost time to cover the remaining test cycles. If no extra resources are available, then planned test cases may have to be cancelled. What is important is to coordinate with the other processes in development.

Agree achievable Deliverables and Milestones at the outset

All test team deliverables, milestones and time frames should be identified and agreed at the outset of the project.

Deliverables that the test teams require include the following:

- User Requirements
- Functional Specification
- Technical Specification
- Program Specification
- Proposed code delivery schedules
- Software to test
- Environments to run the test in
- Details of systems configuration and version
- Backup procedure - in place and working

The test teams will need to provide the following deliverables to the project:

- Functional Hierarchy
- Risk Assessment of the SUT
- Test Specifications
- Test Scripts
- Proposed testing schedules
- Testing progress reports
- Test logs
- Incident log
- Details of incidents
- Exit criteria
- Test completion report

Tasks of the Tester

The various tasks and activities that testers might perform include:

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

- Review tests developed by others.

People who work on test analysis, test design, specific test types or test automation may be specialists in these roles. 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 be business experts and users, and testers for operational acceptance testing would be operators.

5.2. Test planning and estimation

5.2.1. Test planning

Test Plan

> *"A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and test measurement techniques to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process."*

Before planning the tests the following things should be in place:

- Quality assurance plan
- Test strategy
- Test concept

Quality Assurance Plan

Testing should not be the only quality assurance measure used, but should be used in connection with other QA measures. The test plan must be in accord with the quality assurance plan of the project.

The IEEE Standard for Software Quality Assurance Plans (730-2002) establishes required minimum contents for Software Quality Assurance plans. The plan is directed toward the development and maintenance of software. The orientation is toward delineating all of the planned and systematic actions on a particular project that would provide adequate confidence that the software product conforms to established technical requirements.

Test Strategy

Testing should be planned and conducted systematically. One of the ways of ensuring that this is done is to have a fully documented test strategy. The test strategy contains the test requirements for one or more projects within an organization and is based on the organization's test policy.

One of the main purposes of the test strategy is to highlight the risks involved in a project and to identify the testing that will help to address and mitigate those risks. Thus, a test strategy will normally consist of two main parts:

- an explanation of the risks which should be covered by software testing

- the testing which will address those risks

The test strategy may well not be just one physical document, and again, there may be different strategies for different projects, sub-projects or even for different sites in geographically disjointed organizations. Furthermore, the test strategy may not be a document in its own right – it may be produced and included as part of test plans.

Test Concept

A test concept is a document that covers the implementation of the test strategy for a particular project. There are two things that the test concept is *not*:

- It is not a detailed specification of how the tests are to be performed.
- It is not a place where test results are recorded.

Some test organizations combine one or more of these items with the contents of a test plan in order to keep the test documentation in one place. That can work quite well, but it is recommended to use the more modular approach as outlined in the IEEE standard.

The test concept should identify where it conforms to the test strategy. Perhaps more importantly, where it does not conform, it should explain why not.

The planning of test measures is done as early as possible in a software project and is documented in the test concept. According to IEEE 829 the test concept defines the size, procedure, resources and timescales of the foreseen tests.

A test plan is developed based on the concept and represents an illustration of the test concepts laid against the actual course of the project. The plan lays down the chronological order of the tests, contains or references all test cases and assigns test cases to the respective testers. As a "living document" the test plan will be maintained/adapted/changed. On the other hand, the test concept should only be changed in exceptional circumstances.

Sample Test Concept

In order to validate this software in real-world conditions flight testing must be accomplished in which a vehicle equipped with a position/attitude sensor, and equipped with a video camera will fly and record a mission profile in real time. LandForm will then be used to simulate the same flight as viewed from the camera. The imagery will then be quantitatively and qualitatively compared to see how closely the simulated scene matches reality.

The original of this text is in the file http://www.landform.com/downloads/YumaFlightTest.pdf.

High Level Test Plan

The purpose of a high-level test plan is to define the purpose and scope of testing.

The following is a detail of the various elements of a test plan (as per the IEEE 829 Standard for Test Documentation):

Test Plan Identifier

A unique identifier for the test plan.

Introduction

The introduction should give an overview of the test plan. This will typically include a summary of the requirements, what it is hoped testing will achieve

and why it is necessary to test the specified system. In addition, it will reference any external documents or standards that will affect testing - i.e. Quality Assurance documents.

Test Items

This will detail the various physical items that will be used in the tests. These will include the software, hardware and databases. Again any external documents that apply will be referenced.

Features to be tested

This will list all the features of the SUT (SUT) that will be tested under this plan.

Features not to be tested

This will list all the features of the SUT that will not be tested under this plan. Between the test items, features to be tested and features not to be tested we have Scope of the project.

Approach

This describes the approach to testing the SUT. It will be quite high level and should detail what is necessary to perform the testing - the major activities, the testing techniques to be used, the testing tools and aids to be used, any constraints to testing and any support required (environment & staffing). From this information it should be able to ascertain the resources required for testing.

Item Pass / Fail Criteria

Criteria for each test item that define whether that item has passed or failed. This may be expected Vs actual results, certain percentage of tests pass or number of faults remaining (known and estimated).

Suspension & Resumption criteria

Any reasons that would cause testing to be suspended (i.e. non-availability of SUT) and the steps necessary to resume testing.

Test Deliverables

Everything that goes to make up the tests:

- All Documentation (e.g. Specification, Test Plans, Procedures, Reports)
- Code Releases
- Testing Tools (Test Management Tools, Automation Tools, Excel, Word etc.)
- Test Systems (Manual and Automated test cases.)

Testing Tasks

Preparation to perform testing (test case identification, test case design, test data storage and Baseline application), special skills needed (spreadsheet skills, test analysis, automation etc.) and any inter-dependencies.

Environment

Requirements for test environment:

- Hardware & software - PCs, Servers, Routers, SUT, Interfaced Applications, databases
- Configuration - maybe operating systems or middleware to test against
- Facilities - office space, desks, internet access

Responsibilities

Who is responsible - for which activities, for which deliverables and for the environment?

Staffing and Training Needs

The staff required (and their grades), the skills required and any additional training requirements.

Schedules

Timescales, Dates and Milestones.

Risks and Contingencies

What might go wrong and any actions for minimising impact on testing should things go wrong.

Approvals

Who has approved the test plan and names and dates of approval?

5.2.2. Test planning activities

Test planning activities may include:

- Defining the overall approach of testing (the test strategy), including the definition of the test levels and entry and exit criteria.
- Integrating and coordinating the testing activities into the software life cycle activities: acquisition, supply, development, operation and maintenance.
- Making decisions about what to test, what roles will perform the test activities, when and how the test activities should be done, how the test results will be evaluated, and when to stop testing (exit criteria).
- Assigning resources for the different tasks defined.
- Defining the amount, level of detail, structure and templates for the test documentation.
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues.
- Setting the level of detail for test procedures in order to provide enough information to support reproducible test preparation and execution.

Scoping the Testing

It is important to establish what is and what is not to be tested at the beginning of project. Without this information, how can we define clear entry and exit criteria that will enable us to ascertain exactly where we are or even, when we have finished the project?

If we know the scope of the project, then we can also set about determining the resource requirements and any related cost issues.

The scope of the testing to be performed is dependent on the levels of testing that is being performed. Effectively, for any system development or enhancement we must prove that the system is technically correct, functionally correct and also that it satisfies the business requirements of the users.

If the testing that is being undertaken is driven by the Code rather than being driven by the Business / Users' Requirements, i.e. the testers' test what the

developers give them, then there is more likely to be 'Scope Creep' and the finished product may not be what the users actually asked for / wanted.

Test Stages

A new system will have to pass through a number of different testing stages before it goes live. They are listed in the following format as the Acceptance Testing activity should actually begin with the delivery of User Requirements and should not begin with the first delivery of code (as is often the case).

- Component testing
- Integration testing in the small
- Functional system testing
- Non-functional system testing
- Integration testing in the large
- Acceptance testing
- Maintenance testing

Sources of Test Data

Before test scripts can be run, they require pertinent test data to exercise the test condition(s) cross-referenced to them. The test data will be influenced by the type of tests being carried out and by any constraints that the particular project may have.

The process driven approach to the test specification generally describes one transaction or business event, and therefore the test data takes the form of stimuli to exercise relevant test criteria.

By using the data driven approach to testing, the specification describes the business scenario or business thread. The test data required identifies all the different business transactions. For example, if the SUT is for a bank, the business transactions could be the different account types.

The source of the test data needs to be considered. A copy of live data can be used as the test bed, however it must be checked for completeness and any known problems with the data will need to be rectified first. New tables may need to be created for any new functions or new systems; otherwise problems will be encountered, when you commence testing the system. It is also important to consider the issue of confidentiality where live data is being used.

End users / the business will be able to provide details of the necessary make up of the data that will be needed for testing purposes.

A baseline of the test bed is required so that you are able to go back to it when fixes are released and re-testing is required. Once the re-tests have been run successfully, a further baseline can be created which effectively means that there are several baselines throughout the project's life cycle. In order to successfully manage these baselines and also the test data, rigid configuration management / version controls need to be in place.

Finally, if automated test tools are being used in the testing process the format of the date fields need to be considered. Are the dates included within the test bed? If so, when the test is rerun can the system cope with back dated transactions, or will the dates need to be manually adjusted again?

Documentation Requirements

In order to successfully create the functional hierarchy and all the test conditions for the SUT, various levels of documentation must be available to the tester.

For instance, if the tester is undertaking requirements driven testing, details of the system's various functions must be derived from the User Requirement documentation, if it is available. If this documentation is not available to the tester, another way of obtaining these details is to meet the User and discuss their requirements. These meetings with Users should be formal and minuted. The details of the functions / conditions obtained from such meetings should then be cross referenced back to the test cases.

5.2.3. Entry Criteria

Entry Criteria is the set of generic and specific conditions for permitting a process to go
forward with a defined task, e.g. test phase.

Entry criteria are those things that must have successfully completed before this phase of testing can commence. This may be that the previous test stage has completed with no significant defects outstanding or that all software components required for integration testing are available. These criteria will vary from stage to stage and project to project.

The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed 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 may cover the following:
o Test environment availability and readiness
o Test tool readiness in the test environment
o Testable code availability

o Test data availability

5.2.4. Exit criteria

How can we ascertain that a system is ready to go from one stage of testing to another or even ready to go 'live'? This is where defining / adhering to stringent completion or exit criteria is so important. The completion or exit criteria / conditions of a particular testing phase are identified and defined during the planning process for testing the various functional elements of the SUT. Until the completion / exit criteria have been successfully met for a particular testing phase, testing should not progress to the next phase.

Specific test conditions will relate to each separate phase of testing. For instance, the test conditions for User Acceptance Testing will have been derived from the User Requirement documentation.

The entry criteria for a particular stage of testing will include the exit criteria from the previous stage of testing e.g. the exit criteria from Unit Testing will be included within the entry criteria for System Testing. This effectively is a true quality control point.

Test completion or test exit criteria should be used to determine whether the software is ready to be released. The following list is an example of the various kinds of test completion or test exit criteria that can be considered:

1. Has Key Functionality been tested

2. Has test Coverage been achieved
3. Has the Budget been used
4. What is the Defect detection rate
5. Is Performance acceptable
6. Are any defects outstanding

There will never be enough time, money or resource to test everything, but as long as the business critical functions are covered with as much additional testing as it is possible, the software should have significantly fewer defects when it is put live.

5.2.5. Test Estimation

How do we estimate how long we need for testing?

When estimating testing effort we need to work out how long it takes to perform all the tasks described in the Test Plan. The prioritisation of tests is important we need to ensure that the major areas of risk are covered first.

Two approaches of test estimation can be adopted:

- Estimating the testing effort based on metrics of former or similar projects or based on typical values.
- Estimating the tasks by the owner of these tasks or by experts.

In order to give an estimation of testing we need to be able to identify the following:

- The number of tests to be developed and executed.
- The time required to write, test and execute these tests.
- The environment required and the amount of time needed to set it up (both initially and before every test execution).
- The resources and skills required.
- The time required identifying, investigating and logging faults and to then re-testing and regression testing once the faults have been "fixed".

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

- Characteristics of the product: the quality of the specification and other information used for test models (i.e. the test basis), the size of the product, the complexity of the problem domain, the requirements for reliability and security, and the requirements for documentation.
- Characteristics of the development process: the stability of the organization, tools used, test process, skills of the people involved, and time pressure.
- The outcome of testing: the number of defects and the amount of rework required.

Some aspects to consider when estimating:

- The stability of the source code, if the code is still subject to frequent changes that will impact the time to test.
- The responsiveness of the development team when correcting defects will also play a major role.

Always allow time for re-testing!! A common mistake we have noticed in many testing projects is that too little time is allowed for re-testing defects!! The whole idea of testing is to find faults, surely therefore those faults are to be corrected (and re-tested!!).

In addition to the length of time it actually takes to run a test we need to allow some time to set up the environment (PC's, databases etc). We also need to factor in an amount of time for identifying, investigating and logging of faults. Additionally we will need to allow an amount of time for every fault found to be re-tested - this may require the environment to be set and a series of other tests to be executed before the test in which the fault was found can be run. Every new release of the software and every change to the environment require a regression test to be run.

Previous experience with the application, the developers, etc., is a vital aide. Use this in your test estimation & planning. If you know that an application has a complex piece of code that the developers have trouble with and it needs a lot of fine tuning before it is right allow for this in the plan and estimates.

The major difference between estimating testing and other projects is that testing is heavily dependent on other activities. Testing can be easily delayed - you cannot test something until it has been delivered - and testing itself can destroy schedules - finding a number of critical faults will force development to fix them and the tests must be re-run and regression testing must be done before things can move on.

Making accurate estimates

If we have recorded how long a particular task has taken to complete, together with the nature, size and complexity of that task, then we would have a guide to provide us with a reasonably accurate assessment of how long a similar task will take. This will enable us to provide a reasonable estimate of how long we would require completing a particular testing phase or task assuming that the code and environment were stable.

However, it must be stressed that this is only an estimate and from subsequent negotiation, we may not be given the time that we have estimated. Nonetheless, with the supporting evidence that we can provide to the client about how we have arrived at our estimate, we are more likely to have our estimates accepted, or at least not drastically reduced.

Gathering metrics

With any project, there is a fundamental need to gather information that can be used to assess the effectiveness of the testing activities that are being undertaken. The source that is used for gathering these metrics will need to be carefully considered before being chosen.

For example, a typical source document for recording the time taken to complete tasks is the Time Sheet.

Methods of Estimation: the Timesheet

Timesheet for: A.N. Other Beginning of the week							
Activity	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Reviewing Docs							
Script Writing							
Data preparation							
Test Execution							
Office Admin							
Totals							
Weekly Total							

The time sheet is viewed with a large degree of suspicion and is not generally regarded as a reliable source for test metrics. The quality of the information is usually questionable because it is rare for anyone to religiously complete their time sheet on a daily let alone hourly basis. Most people who complete the timesheet do so for the sake of remuneration and do not believe any benefit will accrue from its accurate completion.

The Metrics database

The organisation's management team needs to establish an accurate metrics database to gather information concerning how much time will be required to test each component of the SUT. As data is gathered about the SUT, the manager of the test team should be able to use this information to predict to a reasonable degree of accuracy, the cost of testing the various versions of the new or maintained SUT (assuming that the code and environments are stable).

The Metrics Database

Sample Record	
Reference	Test Script
Reference Name	Set up customer account number 001
Reference Number	010
Class of Item	Writing Script
Attribute	High Level Test Script
Count	45
Time	180
Units of Time	Minutes

The team will have recorded the length of time required to test a particular type / size. The size of the item is calculated by identifying an attribute for example, input field, and then recording the number of occurrences of it.

Where no metrics have previously been recorded or those that have been recorded are of dubious or poor quality, then a start should be made by recording some simple metrics that can be added to later on.

A metrics database is very easy to construct and relatively inexpensive to maintain. However, the information that the database contains, will be both beneficial and influential in establishing realistic estimates for future testing work on similar components in this and other systems.

As the project progresses, there will be other associated data items that may be added to the metrics database to provide a greater source of information to further clarify the testing e.g. complexity, number of defects, rework time etc.

The metrics database will be used to improve, monitor and report on how long testing tasks have taken historically and estimating how long the tasks should take in future. The metrics database is the foundation for accurate predictions of the test effort that will be required for a particular project.

5.2.6. Test Strategy, Test approaches

Approaches or strategies to testing based on when the intensive test design work begins:

- Preventative - tests are designed as early as possible.
- Reactive - test design comes after the software or system has been produced.

Typical approaches or strategies. These can be combined – it is not necessary to use just one.

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

The selection or definition of a strategy always takes into consideration the context.

- Risk of failure of the project, hazards to the product and risks of product failure to humans, the environment and the company.
- Skills and experience of the people in the proposed techniques, tools and methods.
- The objective of the testing endeavor and the mission of the testing team.
- Regulatory aspects, such as external and internal regulations for the development process.
- The nature of the product and the business.

5.3. Test progress monitoring and control

5.3.1. Test progress monitoring

Once testing has started how do we monitor the progress of that effort, and what steps do we take to ensure that the effort progresses as smoothly as possible?

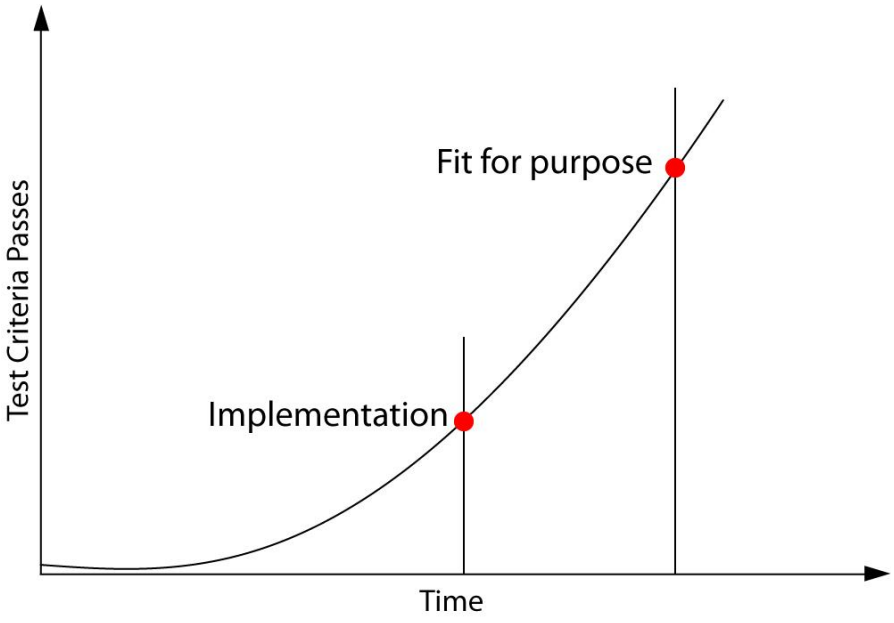
The purpose of test monitoring is to give feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and may be used to measure exit criteria, such as coverage. Measures may also be used to assess progress against the planned schedule and budget.

Common test metrics include:

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

While executing the tests, to assure the validity of the tests more measures can be taken. We can assess the coverage of the tests planned (and allow time to ensure sufficient tests are in place to achieve greater coverage). We can also estimate the amount of time it will take to run the tests.

Fit for Purpose?



5.3.2. Test Reporting

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

- What happened during a period of testing, such as dates when exit criteria were met?
- Analyzed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in tested software.

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

A test summary report shall have the following structure:

- Test summary report identifier;
- Summary;
- Variances;
- Comprehensive assessment;
- Summary of results;
- Evaluation;
- Summary of activities;
- Approvals.

Metrics should be collected during and at the end of a test level in order to assess:

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

5.3.3. Test Control

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

While verifying the application the following measurements can be taken:

- Number of tests run.
- Number of defects raised.
- Number of re-tests run (passed & failed).

If it looks like the testing effort will not be complete in time there are a number of measures that can be taken to get the project back on track, without extending the testing time.

A schedule for anything is generally dictated by 3 entities:

Time, Resource and Scope.

If, as is likely, we cannot change the time we have to look at adjusting one or both of the other entities.

- Re-prioritize tests when an identified risk occurs (e.g. software delivered late).

- Assigning additional resource (either human or otherwise) should always be considered first. Although there is a danger in introducing additional resources late in a project.
- As a last resort a test manager may consider easing the completion or exit criteria.

The Study of Errors by Source

The following two sets of figures illustrate what sort of data can be drawn from simple metrics.

Category	Analysis 1	Analysis 2
1. Requirements	23.5%	8.1%
2. Functionality	24.3%	16.2%
3. Structural	20.9%	25.2%
4. Data	9.6%	22.4%
5. Construction	4.3%	9.9%
6. Integration	5.2%	9.0%
7. Architecture	0.9%	1.7%
8. Test	6.9%	2.8%
9. Unspecified	4.3%	4.7%
Total	**100.0%**	**100.0%**

Test control is maintained by relating the business risks, the business functions, the business priorities, the test scripts and the execution test results together to provide the overall picture of what is happening across the project as a whole. Effective test monitoring will provide the test manager with the exact position with regards to the testing activities that were planned / have been completed at any moment in time. With this level of information available, the test manager will be able to deduce:

- Which test case(s) I am unable to run due to test failures?
- Which test case(s) I must re-run to re-create the test scenario?
- Which test case(s) I am still able to run following the test failure?

The management of the re-testing of the fixes for defects will be considerably easier with all the relevant information readily available. The report that the test manager provides to the client's management team, should identify the current progress that is being achieved against the planned progress and it should also provide details for any area where there is a great variance. The report must also identify the time and resources that will be needed to complete the task of testing for the SUT.

To accomplish this, the test manager will need to know the following information:

- What types of tests have been successfully completed?
- How long did it take to do run the tests?
- What types of tests are still to be run?
- How many defects have been discovered so far?

- How serious are the defects that have been found so far?
- The length and complexity of the tests still to be run.
- Any known / expected delays that will affect the delivery of the code.
- The dependencies for the completion of testing.

To accumulate statistics of this nature, a small database is required and the metrics database that was described earlier would adequately serve this purpose.

5.4. Configuration Management

What is 'Configuration Management'?

> *"A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."*

Traditionally Version Control has been used to control code; it would define a migration path for new or amended code through intermediate environments (allied to test level and status) to the Production environment. However, with the changes in computing from mainframes to midrange, PCs and networks the environments became more complex than ever before. In order to control all the items necessary a myriad of Version Control systems would be required and this becomes unworkable. Something more central was required and the answer is the next level to Version Control, Configuration Management.

Typical Symptoms of Poor CM

With the correct level of configuration management in place, a project can flow through the SDLC without any major problems / glitches. Version control, test case dependency tracking and change control will all operate effectively and everybody will know where within the SDLC they are.

However, poor configuration management can cause many problems and can be identified by some obvious symptoms including:

- **Constantly exceeding the project's deadlines**

  To test any release of efficiently, there must be co-ordination between the code, the requirements, the test material, the test environment and the code libraries. If there is bad or worse still, no co-ordination between the various factors, then deadlines will inevitably be affected / exceeded.

- **Lack of repeatability**

  A lack of repeatability often stems from not knowing how it was done in the first place. With little or no configuration management in place, each software release is a haphazard process and it is impossible for anyone to say with any degree of certainty, whether the new build in anyway resembles the previous build. This can lead to many problems including unstable software because no one is certain what versions of each component can successfully interact with each other.

- **Source and object code mismatches**

  This is a typical problem highlighting poor or non-existent version control. The version of the software that is running in the test environment must be

correctly matched against the version of source code in the program library. When faults occur within the testing environment it is imperative that the developers fix the right version of the code.

Many systems today rely heavily on the use of 'Dynamic Link Libraries' (DLLs). If the wrong version is installed, the application will invariably not run.

For the tester, configuration management helps to uniquely identify (and to reproduce) the tested item, test documents, the tests and the test harness.

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

Configuration Activities

The four activities of configuration management are as follows:

Configuration Identification

"An element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation."

Prior to the commencement of the project, it is necessary to identify, name, and describe the documented physical and functional characteristics of the code, the specifications, the design, and data elements to be controlled for the duration of the project.

This relates to all items that are going to be subjected to configuration management. Everything must be identified and have all details, including versions recorded.

Configuration Control

"An element of configuration management, consisting of the evaluation, co-ordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification."

This relates to the maintenance of the configuration items i.e. it records the changes to the various items over time. As mentioned earlier, it is imperative to know what versions of each component make up a particular build and by keeping strict configuration control records for a project; it is an easy tasks to identify each version of each component.

Status Accounting

"An element of configuration management, consisting of the recording and reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of the approved changes."

This relates to recording / tracking the status of the configuration items e.g. incidents and change requests. There are various categories that can be used to record / track the status of items e.g. initial approved version, status of requested changes, implementation status of approved changes etc.

Configuration Auditing

"The function to check on the contents of libraries of configuration items, e.g. for standards compliance."

This is to ensure that the configuration items reflect those that were defined in the requirement documentation. It is used to determine to what extent the actual configuration item reflects the physical and functional characteristics as defined in the requirement documentation.

Configuration Management covers the processes used to control, co-ordinate, and track: code, requirements, documentation, problems, change requests, designs, tools / compilers / libraries / patches, changes made to them, and who makes the changes.

Software systems are made up of the following: a large amount of code (typically divided into a large number of units) and the environment (PC's, printers, servers, networks etc). In typical commercial software system there may be thousands of software units and thousands of environment items.

A typical system will be released to the testers in a series of "builds". Different builds may have different features enabled or disabled and may include fixes to faults identified in previous builds. Configuration Management enables companies to track the various units and environment set-ups that go to make each build. From this it should be possible to identify which features should have been working in a particular build, which version of which software units were used to make the build and how the environment was configured. This information should enable the company to identify the changes made between builds and to narrow down the likely cause should a "new" fault occur in a new build.

Example

As an analogy - let's look at the components that make up a car.

Simplistically speaking all cars have the same general components whether we are looking at a Lada or a Lancia, they all have, for example a gear-box, a clutch and engine, alternators, batteries, fuel injectors radiators etc.,

However while the components may be the same generally, you won't be able to run a Lancia if a Lada clutch is fitted.

Even in different 'versions' of essentially the same car the parts may not be interchangeable.

Most car mechanics know what parts and what versions of parts go into which model of the car.

In IT parlance this information is known and documented through Configuration Management.

Configuration Management Web Sites

For further information on Configuration Management, have a look at the following web sites:

http://www.perforce.com

http://www.fpcl.co.uk

http://www.dis.port.ac.uk/~allangw/papers/pub97d.htm

http://www.rational.com/products/clearcase/index.jsp

http://www.bcs-cmsg.org.uk/

http://www.cmtoday.com

http://burks.bton.ac.uk/burks/foldoc/76/23.htm

http://www.merant.com/products/pvcs/dimensions/index.asp?source=goto

5.5. Risk and testing

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

5.5.1. Project risks

Project risks relate to the project's capability to deliver its objectives:

- Supplier issues: failure of a third party, contractual issues.
- Organizational factors:
  - skill and staff shortages;
  - personal and training issues;
  - political issues, (testers have problems communicating their needs and test results; failure to follow up on information found in testing and reviews).
  - improper attitude toward or expectations of testing (e.g. not appreciating the value of finding defects during testing).
- Technical issues:
  - problems in defining the right requirements;
  - the extent that requirements can be met given existing constraints;
  - the quality of the design, code and tests.

When analyzing, managing and mitigating these risks, the test manager is following well established project management principles. The 'Standard for Software Test Documentation' (IEEE 829) outline for test plans requires risks and contingencies to be stated.

5.5.2. Product Risks

Product risks are potential failure areas in the software or system, as they are a risk to the quality of the product:

- Error-prone software delivered.
- The software/hardware could cause harm to an individual or company.
- Poor software characteristics (e.g. functionality, security, reliability, usability and performance).
- Software does not perform its intended functions.

A **risk-based approach to testing** is a proactive opportunity to reduce the levels of product risk, starting in the initial stages of a project. The risks identified may be used to:

- Determine the test techniques to be used.
- Determine the extent of testing.
- Prioritize testing in order to find the critical defects as early as possible.
- Determine necessary non-testing activities that can reduce risk (e.g. train staff).

Risk management activities provide a disciplined approach to:

- Assess (and reassess on a regular basis) what can go wrong (risks).
- Determine what risks are important to deal with.
- Implement actions to deal with those risks.
- Identify new risks (with the help of testing).

5.5.3. Risk analysis

To maximise the effectiveness of the overall testing process, business risk analysis is undertaken, so that effort and resource can be allocated to testing the most 'business critical' functions of the SUT.

Business risk relates to the underlying cost of a failure of the system or function within the system. The complexity of the function and the perceived likelihood of failure are taken into account when assigning risk factors. Producing testing progress information categorised by risk factor assists senior project management to determine the associated risks of not performing certain tests or failing to implement parts of the system.

A risk factor should be allocated to each function in order to differentiate between:

Critical functions - these must be fully tested and available as soon as the changes go live. The cost to the business is high if these functions are unavailable for any reason and an alternative work around is sometimes impossible or very difficult.

Critical functions include:

- Reports / enquiries where the transaction is used to make critical management decisions
- High-risk transactions in real-time e.g. the processing of an entry on a bank account

Required functions - are not absolutely critical to the business. It is usually possible to find adequate methods to 'work around' these problems using other mechanisms, e.g. existing systems, manual procedures etc.

This class of functions includes:

- Maintaining data where errors are more likely to cause inconvenience rather than serious problems / concerns
- Transaction or batch updates where errors could lead to small and acceptable discrepancies, which can be readily identified and corrected using alternate processes (including manual intervention)
- Reports that are required by the management on a daily basis but are not needed to make critical decisions

Nice to have' functions – these functions **can** easily be worked around or left out all together. These include: functions maintaining non-critical data such as source information or static data that rarely changes; reports or enquiries where no information is used to make critical management decisions.

Remember that the purpose of assigning risk to each function is to build the optimum risk reduction profile where the functionality that is most critical to the system is coded / tested first.

Other risks that may need to be considered when planning include:

- Schedule risk – Not having enough time to complete the testing
- Testing risk – Inadequate testing
- Cost risk – Not having enough money to complete the testing
- Program risk – Poor quality code

5.6. Incident Management

Since one of the objectives of testing is to find defects, the discrepancies between actual and expected outcomes need to be logged as incidents. Incidents should be tracked from discovery and classification to correction and confirmation of the

solution. In order to manage all incidents to completion, an organization should establish a process and rules for classification.

What is an Incident?

"Any event occurring during testing that requires investigation."

An INCIDENT is an unplanned event that occurs during testing and demands investigation and / or correction to allow testing to proceed as planned. This is intended to be a dispassionate word to describe a problem with the system - bug is an emotive word, it implies that a programmer has made a mistake. The term 'incident management' includes in itself the term 'defect management' or 'problem management'.

Incidents can be raised during development, review, testing or use of a software product. They may be raised for issues in code or the working system, or in any type of documentation including development documents, test documents or user information such as "Help" or installation guides. (Suggested enhancements to the system are not normally logged as Incidents - as they are not a fault.)

The Impact of an Incident

An incident needs to be viewed in terms of its' impact upon testing.

As the final system release date draws nearer the definitions of priorities will change. Only those things that are perceived a significant risk to the business (i.e. the applications critical path cannot be used) will be deemed an urgent priority.

Incidents are often measured by their Severity (the impact of the failure) and Priority (the urgency to fix the fault).

Prioritising Incidents

Incidents are prioritised in terms of their impact, initially on the actual testing of the SUT, but as testing progresses the potential effect of such an incident on live users is borne in mind. This reflects the various stages of testing used as the system moves towards live use.

The incidents should also be analysed in detail (causal analysis) to identify the underlying cause, and time permitting fixes should be targeted at the root cause of the problem rather than fixing the manifestation of the problem.

Such an analysis should indicate how the system is evolving and the true progress of the system towards production readiness can be better assessed.

There is a temptation to over complicate the recording of incidents, particularly where the priority of the incident is concerned.

However practical experience suggests that three levels are sufficient.

Recording Incidents

Incident reports have the following objectives:

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

When an incident has been identified and analysed (to ensure that it is a fault) sufficient information should then be recorded to enable the fault to be rectified.

- Test ID, System ID, Testers ID.
- Expected and Actual results.
- Environment in use.
- Severity of the incident.
- Priority of the incident.
- Any other information.
- Date and time of test execution.
- System builds information.

Enough information about the incident must be provided so that the recipient of the incident is able to quickly reproduce the fault (if necessary) and is able to identify it and fix it.

Details of the incident report may include:

- Date the incident was discovered.
- Identification or configuration item of the software or system.
- Software or system life cycle process in which the incident was observed.
- Description of the anomaly to enable resolution.
- Degree of impact on stakeholder(s) interests.
- Status of the incident (e.g. open, deferred, duplicate, waiting to be fixed, fixed awaiting confirmation test or closed).
- Conclusions and recommendations.
- Global issues, such as other areas that may be affected by a change resulting from the incident.
- Change history, such as the sequence of actions taken by project team members with respect to the incident to isolate, repair and confirm it as fixed.

The structure of an incident report is covered in the 'Standard for Software Test Documentation' (IEEE 829) and is called an anomaly report.

Tracking incidents

Once raised the incident should be tracked and monitored. Progress and the responsiveness of the development team should be monitored.

The following is a very simplified life cycle of an incident. These can vary, but in essence the core remains very much the same.

- When first raised an incident is OPEN.
- It then passes to development to fix - WITH DEVELOPMENT.
- Development may question the incident - PENDING.
- Or fix the problem and return it for - RETEST.
- Should the retest prove successful the defect is CLOSED.

The various priorities and statuses should be monitored and reported on regularly.

By gathering and presenting these metrics the progress of the application towards production worthiness can be measured in quantifiable terms.

Tracking and Analysis

The history of any incident that occurs during the testing process must be kept in order to provide a complete record of what problems were encountered and identify the current status of the incident. Typical entry and exit criteria can include the number of incidents that occurred at various severity levels and for this reason, it is imperative to have agreed corporate level standardised priority codes prior to the testing commencing.

Again, regular reports to the management are required in order to manage the number / severity of the incidents and the time that is being taken to achieve fixes.

Incident history logs must be established at each stage of the incident resolution process so that it can be tracked and for future audit purposes. This will also provide a means of formally documenting the incidents and provide details of the particular departments who own them at any particular point in time.

5.7. Summary

Organisation

The organisational structure for testing should take into account the following:

- Identify & assign test responsibilities
- Determine the requisite skill sets
- Requisition resource (internal and external)
- Decide reporting lines
- Agree deliverables and milestones

The composition of the test team varies according to the different testing undertaken and the size of the project.

Organisations may have different testing structures:

- Testing may be the developer's responsibility.
- Or may be the team's responsibility (buddy testing),
- Or one person on the team is the tester,
- Or there is a dedicated test team (who do no development),
- Or there are internal test consultants providing advice to projects,
- Or a separate organisation does the testing.

Test Planning

IEEE definition of test plan: "A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning."

The 16 sections in an IEEE test plan:

- **Test plan identifier**

Specify the unique identifier assigned to this test plan.

- **Introduction**

  Summarise the software items and software features to be tested. The need for each item and its' history may be included.

- **Test items**

  Identify the test items including their version / revision level.

- **Features to be tested**

  Identify all software features and combinations of software features to be tested.

- **Features not to be tested**

  Identify all features and significant combinations of features that will not be tested and the reasons.

- **Approach to testing**

  Describe the overall approach to testing.

- **Item pass / fail criteria**

  Specify the criteria to be used to determine whether each test item has passed or failed testing.

- **Suspension and resumption criteria**

  Specify the criteria used to suspend all or a portion of the testing activity on the test items associated with this plan. Specify the testing activities that must be repeated, when testing is resumed.

- **Test deliverables**

  Identify the deliverable documents.

- **Testing tasks**

  Identify the set of tasks necessary to prepare for and perform testing.

- **Environment**

  Specify both the necessary and desired properties of the test environment.

- **Responsibilities**

  Identify the groups responsible for managing, designing, preparing, executing, witnessing, checking, and resolving.

- **Staff and training needs**

  Specify test staffing needs by skill level. Identify training options for providing necessary skills.

- **Schedule**

  Include test milestones identified in the software project schedule as well as all item transmittal events.

- **Risks and contingencies**

  Identify the high-risk assumptions of the test plan. Specify contingency plans for each.

- **Approvals**

  Specify the names and titles of all persons who must approve this plan.

Configuration Management

Configuration management (CM) is the detailed recording and updating of information that describes an enterprise's computer systems and networks, including all hardware and software components. Such information typically includes the versions and updates that have been applied to installed software packages and the locations and network addresses of hardware devices.

Typical Symptoms of poor CM:

- Missed deadlines
- Too many faults are found in the 'live' system
- High turnaround time for bug fixes
- Lack of repeatability in development
- Mismatches in Source code and object code

Configuration Management Terms

Configuration identification

Requires that all *configuration items* (CI) and their versions in the test system are known.

Configuration control

Is maintenance of the CIs in a library and maintenance of records on how CIs change over time?

Status accounting

Is the function recording and tracking problem reports, change requests, etc?

Configuration auditing

Is the function to check on the contents of libraries, etc. for standards compliance, for instance?

Test Estimation, Monitoring and Control

Estimating the length of a testing project, monitoring its performance and taking steps should it move off schedule.

Test Estimation

The effort required to perform activities specified in the high-level test plan must be calculated in advance and that rework must be planned for.

How do we estimate how long we need for testing?

Estimation – factors to consider.

- Risk of failure
- Complexity or code / solution.
- Critically of functions.
- Coverage.
- Stability of SUT.
- Availability of resources, data, environment.

- Time required for re-testing and regression testing.

Test Monitoring

The current state of testing can be measured (e.g. number of tests run, tests passed / failed, incidents raised and fixed, retests, etc.) so that we can determine whether or not the project is running to schedule.

Monitoring – how to keep track.

Test preparation.

- Estimate number of tests needed.
- Estimate time to prepare them.
- Track percentage of tests prepared.

Test execution:

- Coverage achieved to date (number of functions, tests run).
- Estimate time to run test suite.

Whilst testing you can monitor:

- Number of tests run and passed & failed.
- Defects raised.
- Re-tests run and passed and failed.

Any deviation from schedule should be flagged.

Test Control

Control is maintained by relating business risks, functional areas, priorities, test scripts and execution results together to form the 'big picture'.

If it is discovered that there has been some slippage on the schedule, the following methods can be utilised to bring the project back on schedule:

- Assign more resources / re-allocate resources.
- Adjust test schedule.
- Arrange for extra test environments.
- Refine completion criteria.

Incident Management

What is an Incident?

"An incident is any significant, unplanned event that occurs during testing that requires subsequent investigation and /or correction."

What might an incident be raised against?

Documents, code, tests, SUT, environment

Incidents and the Test Process

Incidents will affect the test process. Serious incidents may actually halt the process completely while less serious incidents may go virtually unnoticed. It is therefore important to identify both where the incident occurred and how serious it is.

Monitoring and Analysis of Incidents

Incidents need to be logging, classified and given a status as they are encountered. This will enable us to track them to ensure they are resolved and then analysed once the project is complete.

The following is a very simple and simplified life cycle of an incident. These can vary, but in essence the core remains very much the same.

- When first raised an incident is OPEN.
- It then passes to development to fix - WITH DEVELOPMENT.
- Development may question the incident - PENDING.
- Or fix the problem and return it for - RETEST.
- Should the retest prove successful the defect is CLOSED.

References

5.1.1. – Black, 2001, Hetzel, 1988

5.1.2. - Black, 2001, Hetzel, 1988

5.2.5. - Black, 2001, Craig, 2002, IEEE 829, Kaner, 2002

5.3.3. - Black, 2001, Craig, 2002, Hetzel, 1988, IEEE 829

5.4. - Craig, 2002

5.5.2. – Black, 2001, IEEE 829

5.6. - Black, 2001, IEEE 829

6. Tool support for testing

6.1. Types of test tools

6.1.1. Understanding the Meaning and Purpose of Tool Support for Testing

Test tools are software products or techniques that support testing activities. Testing tools cover a wide range of activities and are applicable for use in all phases of the development life cycle.

Test tools can be used for one or more activities that support testing. These include:
1. Tools that are directly used in testing such as test execution tools, test data generation tools and result comparison tools
2. Tools that help in managing the testing process such as those used to manage tests, test results, data, requirements, incidents, defects, etc., and for reporting and monitoring test execution
3. Tools that are used in reconnaissance, or, in simple terms: exploration (e.g., tools that monitor file activity for an application)
4. Any tool that aids in testing (a spreadsheet is also a test tool in this meaning)

They can improve the efficiency of testing activities by automating repetitive tasks. Testing tools can also improve the reliability of testing by, for example, automating large data comparisons or simulating behaviour.

Tool support for testing can have one or more of the following purposes depending on the context:

- Improve the efficiency of test activities by automating repetitive tasks or supporting manual test activities like test planning, test design, test reporting and monitoring
- Automate activities that require significant resources when done manually (e.g., static testing)
- Automate activities that can not be executed manually (e.g., large scale performance testing of client-server applications)
- Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

6.1.2. Test tools classification

Testing tools cover a wide range of activities and are applicable for use in all phases of the development life cycle. Some of the tools are software products, some are techniques; some techniques are manual and some automated; some perform static tests, others perform dynamic tests; some evaluate the system structure, and others, the system function.

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. Some commercial tools offer support for only one type of activity; other commercial tool vendors offer suites or families of tools that provide support for many or all of these activities.

Some types of test tool can be intrusive in that the tool itself can affect the actual outcome of the test. For example, the actual timing may be different depending on how you measure it with different performance tools, or you may get a different measure of code coverage depending on which coverage tool you use. The consequence of intrusive tools is called the probe effect.

The skill required to use the tools and the cost of executing the tools vary significantly. Some of the skills are highly technical and involve an in-depth knowledge of computer programming and the system being tested. Other tools are general in nature and are useful to almost anyone with testing responsibilities. The cost of some tools only involves a short expenditure of people time, while others must be conducted by a team and make heavy use of computer resources in the test process.

Software Products as Test Tools

A plethora of tools is available on the market. This session looks at the areas where these tools help with testing. These tools are often referred to as **CAST tools** - Computer Aided Software Testing. The term is borrowed from the term CASE (Computer Aided Software Engineering)

> **Note**. Many families of tools from a single vendor encompass many of these topics. The full suite of Mercury products for example will cover: Data Preparation, Character based test running, GUI Testing, Performance Testing, Comparison and Test Management. Similar offerings from other vendors also offer similar coverage.

Techniques as Test Tools

Summarised in the table below are the more common testing techniques:

	Tool name	Description and use
1	**Acceptance Test Criteria**	Provides the standards that must be achieved for the system to be acceptable to the user.
2	**Boundary Value Analysis**	Divides system top down into logical segments and then limits testing within the boundaries of each segment.
3	**Cause-Effect Graphing**	Limits the number of test transactions by determining which of the number of variable conditions pose minimal risk based on system actions.
4	**Checklist**	Provides a series of questions designed to probe potential system problem areas.
5	**Code Comparison**	Compares two versions of the same program in order to identify differences between the two versions.
6	**Compiler-based Analysis**	Detects errors during the program compilation process.
7	**Complexity-based Metric**	Uses relationships to demonstrate the degree of system processing complexity provided by the test process.
8	**Confirmation/ Examination**	Verifies that a condition has or has not occurred.
9	**Control Flow Analysis**	Identifies processing inconsistencies such as routines with no entry point, potentially unending loops, branches into the middle of a routine, etc.
10	**Correctness Proof**	Requires a proof hypothesis to be defined and then used to evaluate the correctness of the system.
11	**Coverage-based Metric Testing**	Uses relationships to demonstrate the degree of system processing complexity provided by the test process.
12	**Data Dictionary**	Generates test data to verify data validation programs based on the data contained in the dictionary.
13	**Data Flow Analysis**	Identifies defined data not used and used data that is not defined.
14	**Design-based Functional Testing**	Evaluates functions attributable to the design process as opposed to design requirements; for example, capability may be a design process.
15	**Design Reviews**	Requires reviews at predetermined points throughout systems development in order to examine progress and ensure the development process is followed.
16	**Desk Checking**	Provides an evaluation by programmer or analyst of the propriety of program logic after the program is coded or the system is designed.
17	**Disaster Test**	Simulates an operational or systems failure to determine if the system can be correctly recovered after the failure.
18	**Equivalence checking**	Prove by exhaustive testing that two expressions are equivalent. Also Equivalence testing.
19	**Error Guessing**	Relies on the experience of testers and the organization's history of problems to create test transactions that have a high probability of detecting an error.
20	**Executable Specs**	Provides a high-level interpretation of the system specs in order to create the response to test data. Interpretation of expected software packages requires

		system specs to be written in a high-level language.
21	**Exhaustive Testing**	Attempts to create a test transaction for every possible condition and every path in the program.
22	**Fact Finding**	Performs those steps necessary to obtain facts to support the test process.
23	**Flowchart**	Pictorially represents computer systems logic and data flow.
24	**Inspections**	Requires a step-by-step explanation of the product with each step checked against a predetermined list of criteria.
25	**Instrumentation**	Measures the functioning of a system structure by using counters and other monitoring instruments.
26	**Integrated Test Facility**	Permits the integration of test data in a production environment to enable testing to run during production processing.
27	**Mapping**	Identifies which part of a program is exercised during a test and at what frequency.
28	**Modelling**	Simulates the functioning of the environment or system structure in order to determine how efficiently the proposed system solution will function.
29	**Parallel Operation**	Verifies that the old and new version of the application system produces equal or reconcilable results.
30	**Parallel Simulation**	Approximates the expected results of processing by simulating the process to determine if test results are reasonable.
31	**Peer Review**	Provides an assessment by peers of the efficiency, style, adherence to standards, etc. of the product which is designed to improve the quality of the product.
32	**Risk Matrix**	Produces a matrix showing the relationship between system risk, the segment of the system where the risk occurs, and the presence or absence of controls to reduce that risk.
33	**SCARF (System Control Audit Review File)**	Builds a history of potential problems in order to compare problems in a single unit over a period of time and/or compares like units.
34	**Scoring**	Identifies areas in the application that require testing, through the rating of criteria that have been shown to correlate to problems.
35	**Snapshot**	Shows the content of computer storage at predetermined points during processing.
36	**Symbolic Execution**	Identifies processing paths by testing the programs with symbolic rather than actual test data.
37	**System Logs**	Provides an audit trail of monitored events occurring in the environment area controlled by system software.
38	**Test Data**	Creates transactions for use in determining the functioning of a computer system.
39	**Test Data Generator**	Provides test transactions based on the parameters that need to be tested.
40	**Tracing**	Follows and lists the flow of processing and database searches.
41	**Utility Program**	Analyses and prints the result of a test through the use of a general-purpose program (i.e., utility program).

42 Volume Testing	Identifies system restriction (e.g., internal table size) and then creates a large volume of transactions designed that exceed those limits.
43 Walkthroughs	Leads a test team through a manual simulation of the product using test transactions.

6.1.3. Tool support for management of testing and tests

Test management tools

Test management tools are designed to centrally manage and store test assets – scripts, plans, results, incident reports etc. There are a wide range of test management tools on the market - all of which do different things and have different focuses.

Test management tools are useful throughout the SDLC. For instance, it is possible to buy Test Management tools to help support test planning, test monitoring, incident management etc.

It is very common for test management tools to be integrated with (or provide an interface to) other testing tools. This is extremely helpful if you are running automated tests because it is possible to execute the test cases from the test management tool.

Characteristics of test management tools include:

- Support for the management of tests and the testing activities carried out.
- Interfaces to test execution tools, defect tracking tools and requirement management tools.
- Independent version control or interface with an external configuration management tool.
- Support for traceability of tests, test results and incidents to source documents, such as requirement specifications.
- Logging of test results and generation of progress reports.
- Quantitative analysis (metrics) related to the tests (e.g. tests run and tests passed) and the test object (e.g. incidents raised), in order to give information about the test object, and to control and improve the test process.

Requirements Management Tools

> *"A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to pre-defined requirements rules."*

Requirements Testing Tools

These tools can read a specification and check its conformance to standards. They are not able to examine a specification and determine whether a requirement is valid.

Often rather than use such a tool the requirements are reviewed in workshops where users are asked to review the requirements documentation to ensure it satisfies their needs.

Requirements testing tools are a relatively new type of tool that can help with the task of analysing requirements. These tools can work on requirement specifications written in a formal structured language or even normal English. Although they are not able to help to validate user's requirements (i.e. tell you if the requirements are what the end user actually wants), they can help with verifying the requirements (i.e. checking conformance to standards for requirements specifications).

Many of today's word processing applications can be seen to be very basic requirement-testing tools, since one of the many functions of these applications is to check both spelling and grammar, as the user types words into the document.

Ambiguity in a requirement specification has often lead to serious faults in the system that has been delivered and these ambiguities are sometimes caused by poor grammar within the requirement specification. Proper requirements testing tools offer a much richer set of functionality than mere grammar checking although this too is one of their functions.

For instance, they can check for consistent use of specific terms throughout a specification and derive a list of possible test conditions for acceptance testing but this should only be used as a starting point.

Sometimes, these tools can lead to false confidence because it is easy to assume that if the tool does not find anything wrong with a requirement specification, then it is perfect. These tools do require manual intervention, they are not automatic and they certainly cannot correct all the (potential) faults that they find.

Perhaps the most obvious pitfall is that the requirements have to be written down. Many organisations fail to produce a complete requirements specification and for them this type of tool will have limited value.

Incident Management Tools

> *"A tool that facilitates the recording and status tracking of incidents found during testing. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of incidents and provide reporting facilities."*

Incident management tools support management of incident reports in ways that include:

- Facilitating their prioritization.
- Assignment of actions to people (e.g. fix or confirmation test).
- Attribution of status (e.g. rejected, ready to be tested or deferred to next release).

These tools enable the progress of incidents to be monitored over time, often provide support for statistical analysis and provide reports about incidents. They are also known as defect tracking tools.

Configuration management tools

Configuration management tools are not strictly testing tools, but are typically necessary to keep track of different versions and builds of the software and tests.

Configuration management tools can be used for:

- Identification of software configuration item
- Definition baselines
- Archiving (check-in/check-out)
- Version Control

- Change Management and history
- Build and Release Management
- Status accounting
- Distributed development
- Access Control

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

6.1.4. Tool support for static testing

Review process support tools

Review process support tools may store information about review processes, store and communicate review comments, report on defects and effort, manage references to review rules and/or checklists and keep track of traceability between documents and source code. They may also provide aid for online reviews, which is useful if the team is geographically dispersed.

Static Analysis Tools

> **static analyzer:** *A tool that carries out static analysis.*
>
> **static code analyzer:** *A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies.*

Static analysis tools support developers, testers and quality assurance personnel in finding defects before dynamic testing. Their major purposes include:

- The enforcement of coding standards.
- The analysis of structures and dependencies (e.g. linked web pages).
- Aiding in understanding the code.

Static analysis tools provide information about the quality of the software by examining the code, rather than by running test cases through the code. They are able to identify basic coding errors i.e. unreachable code, eternal loops, incorrect syntax and variables being used before they have been declared. These and many more potential faults can be difficult to see when reading source code but can be picked up within seconds by a static analysis tool.

Static analysis tools usually give objective measurements of various characteristics of the software, such as McCabe's Cyclomatic complexity measure Halstead metrics and many more other quality metrics.

In simple terms, they are a type of super compiler that will highlight a much wider range of real or potential problems than compilers do. Static analysis tools detect all of a particular type of fault much more effectively than can be achieved by any other means.

Although static analysis tools are an extremely valuable type of testing tool, they are not used by many organisations. The pitfalls are more psychological than real, for example, a static analysis tool may highlight something that is not going to cause a failure of the software. This is because it is a static view of the software.

Modelling tools

Modelling tools are able to validate models of the software. For example, a database model checker may find defects and inconsistencies in the data model; other modelling tools may find defects in a state model or an object model. These tools can often aid in generating some test cases based on the model (see also Test design tools below).

The major benefit of static analysis tools and modelling tools is the cost effectiveness of finding more defects at an earlier time in the development process. As a result, the development process may accelerate and improve by having less rework.

6.1.5. Tool support for test specification

Test design tools

> **test design tool:** *A tool that supports the test design activity by generating test inputs from a specification that may be held in a CASE tool repository, e.g. requirements management tool, or from specified test conditions held in the tool itself.*

These tools assist with the generation of test cases. Test design tools generate test inputs or the actual tests from requirements, from a graphical user interface, from design models (state, data or object) or from code. This type of tool may generate expected outcomes as well (i.e. may use a test oracle).

When generating tests from requirements, the specification has to in a formal language that the test design tool understands or for some tools a CASE (Computer Aided Software Engineering) tool can actually hold it. A CASE tool captures much of the information required by the test design tool, as the system is being designed. It therefore saves the need to re-specify the design information in a different format just for the use of the test design tool.

Where a test design tool uses the user interface of an application, the user interface has to be implemented first, prior to being able to use the test design tools. It is also a fairly restricted set of test inputs that can be generated in this way, since these tools concentrate on testing the user interface rather than the underlying functionality of the software. However, this can still be very useful.

When it is the source code that is to be used to generate test inputs, they are useful for checking that the code does what the code does. However, it is often more useful to check that the code does what the code is supposed to do and this is what tests based on the system specification actually do.

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

Other tools in this category can aid in supporting the generation of tests by providing structured templates, sometimes called a test frame, that generate tests or test stubs, and thus speed up the test design process.

Test data preparation tools

> *"A type of test tool that enables data to be selected from existing databases or created, generated, manipulated and edited for use in testing."*

These tools help you build and prepare data for testing. The quality of testing achieved is highly dependant on the QUALITY rather than the QUANTITY of data used.

There are a number of tools available in the market place to aid with test data preparation.

Test data can come from many sources - A subset of production data can be used, the subset can be gained by taking either a random sample of production or a selective extract. Data can be built or created specifically for testing. In an ideal world we should always use the application under test to create its own test data, but this is not always possible or practical.

A data preparation tool makes it much easier to generate large volumes of data (as required for volume, performance and stress testing for example) when it is needed. This makes the process more manageable, as the data can be regenerated whenever it is required. On the downside, the technical set up for complex test data may be rather difficult / time consuming or at least very tedious.

6.1.6. Tool support for test execution and logging

Test execution tools

Test execution tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language. The scripting language makes it possible to manipulate the tests with limited effort, for example, to repeat the test with different data or to test a different part of the system with similar steps. Generally these tools include dynamic comparison features and provide a test log for each test run.

Test execution tools can also be used to record tests, when they may be referred to as capture playback tools. Capturing test inputs during exploratory testing or unscripted testing can be useful in order to reproduce and/or document a test, for example, if a failure occurs.

- **Character Based Test Running Tools**

  These are tools that are designed to execute tests (normally functional system and regression tests) on a character based system - terminals for Mainframes, UNIX and Mid-range systems (System 36/38) and early PC systems.

  Historically these tools were very simple record and playback tools that simply mimicked the data entered by the users and repeated or replayed the process. The results could be compared to previous execution or other predefined results to check the application.

  > **capture/playback tool:** *A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e. replayed). These tools are often used to support automated regression testing.*

  Gradually these tools evolved such that you could create a script using a form of script programming language, and the data, expected results could be held in separate repositories and maintained accordingly.

  This provided the foundations on which many of the GUI tests running tools were built. It may also be considered as a limiting factor for the GUI testing tools that were created based on a Record / Playback approach.

  Test running tools are most often used to automate regression testing and as mentioned earlier, they usually offer some form of capture/replay facility to record the test that is being performed manually, so that it can then replay the same key strokes. The recording of the keystrokes is captured in a test script that can then be edited (or written from scratch) and is then used by the tool to re-perform the test.

Test running tools can be used for test execution during any phase i.e. form unit through to acceptance testing. The benefits include faster test execution and unattended test execution which in turn, reduces the manual effort and permits more tests to be run in less time.

The pitfalls to using test-running tools are enormous and it has been suggested that they have lead to as many as half of all test automation projects to fail in the long term.

The cost of automating a test case is usually far more than the cost of running the same test case manually – it is believed that it is between 2 and 10 times more expensive. Also, the cost of maintaining the automated test cases i.e. updating them to work on new versions of the software can also become an expensive exercise particularly if there are small amendments e.g. renaming of a field in the background or the movement of the field by one pixel, on a regular basis.

- **GUI Test Running Tools**

  There are many tools in the market place that are available as GUI testing tools. Many of these can also perform Character based testing as noted previously. These tools are most often used to automate regression testing.

  You create test scripts, screen definitions, test input data etc. to test the application. The tools simulate mouse movements, button clicks and keyboard inputs and can recognise GUI objects such as windows, edit fields, buttons and other controls.

  Object states and bitmaps can be captured for later comparison Tests, data, test cases and expected results may be held in separate repositories.

Test harness/unit test framework tools

A test harness may facilitate the testing of components or part of a system by simulating the environment in which that test object will run. This may be done either because other components of that environment are not yet available and are replaced by stubs and/or drivers, or simply to provide a predictable and controllable environment in which any faults can be localized to the object under test. These are usually bespoke programs generated by developers to help in the testing process. If they are used in a mature organisation it is quite possible that these harnesses will be considered as 'Test Assets' and subject to Version Control & Configuration Management.

Not all software can be turned into an executable program. For example, a library function that a developer may use as a building block within his or her program needs to be tested separately first, before being used elsewhere. This requires a harness or driver, which is a separate piece of source code that is used to pass test data into the function and then receive the output from it.

When testing at unit level and in the early stages of integration testing, these drivers are usually custom built. However, there are a few commercial tools that can provide a level of support although these are most likely to be language specific.

At the later stages of the SDLC e.g. system or acceptance testing, harnesses (which are also sometimes known as simulators) may be required to simulate hardware systems that are not yet available or cannot be used because the software is not reliable enough.

A framework may be created where part of the code, object, method or function, unit or component can be executed, by calling the object to be tested and/or giving feedback to that object. It can do this by providing artificial means of supplying

input to the test object, and/or by supplying stubs to take output from the object, in place of the real output targets.

Test harness tools can also be used to provide an execution framework in middleware, where languages, operating systems or hardware must be tested together.

They may be called unit test framework tools when they have a particular focus on the component test level. This type of tool aids in executing the component tests in parallel with building the code.

Test comparators

Comparator = test comparator: *A test tool to perform automated test comparison.*

Test comparison: *The process of identifying differences between the actual results produced by the component or system under test and the expected results for a test. Test comparison can be performed during test execution (dynamic comparison) or after test execution.*

Comparison tools are able to examine two or more files and highlight any differences between them. At their simplest they are able to load two text files and show the differences line by line.

For instance, test running tools will usually offer some form of dynamic comparison facilities which will enable the output to the screen during the execution of a test case to be compared with the expected output. However, test running tools are not as good at comparing other types of test outcome e.g. changes to a database and for this task, stand-alone comparison tool should be used.

Comparison tools offer vastly improved speed and accuracy over the manual methods of comparison. These tools will highlight all the differences they find, even ones that you are not interested in e.g. differences in dates / times, unless you are able specify some form of mask or filter to hide these expected differences. Specifying filters / masks may not always be an easy task.

Coverage Measurement Tools

coverage tool: *A tool that provides objective measures of what structural elements, e.g. statements, branches have been exercised by the test suite.*

By monitoring code coverage we are able to see how much (i.e. by percentage) of the code has been "exercised" during testing. Coverage measurement tools can be either intrusive or non-intrusive depending on the measurement techniques used, what is measured and the coding language.

This information is typically used:

- To monitor the state of testing.
- To provide management with information about the level of testing - to enable them to make an educated decision about releasing code.
- To enable testers to create additional tests to cover areas of the code that has not yet been run during testing.
- Aid to development to focus on critical areas.

Coverage tools assess how much of the software under test has been exercised by the set of tests that have been run. The most common way that coverage tools achieve this is by the tool itself running the source code but there are other methods as well.

This requires the tool to insert new instructions within the original program code and these new lines of code then write to a data file recording that the line has

been executed. After one or more test cases have been executed, the tool is then used to examine this newly created data file to determine which parts of the original source code have been executed and (more importantly) which parts have not been executed.

Coverage tools are most commonly used at unit test level by the developers. For instance, branch coverage is often a requirement for testing safety-critical or safety-related systems e.g. the NATS systems (the National Air Traffic control system that recently went live at Swanick).

It is important to set the correct level of coverage measure and this should not just be an arbitrary figure set as a target, without a good understanding of the consequences of doing so. Achieving 100% branch coverage may sound like a good goal but it can be very expensive to actually achieve. It may also not represent the best way of achieving the required level of testing of the SUT.

Security tools

Security tools check for computer viruses and denial of service attacks. A firewall, for example, is not strictly a testing tool, but may be used in security testing. Other security tools stress the system by searching specific vulnerabilities of the system.

6.1.7. Tool support for performance and monitoring

Dynamic Analysis Tools

Dynamic analysis tools monitor the SUT as it is executed. They watch the way the application handles memory, allocating and de-allocating it. This enable testers and developers to see how the SUT is interacting with memory and to highlight any memory address clashes and memory leaks (memory that his not being released by the program when it is finished with). If a program does contain a memory leak, it will eventually end up owning all of memory and nothing will be able to be run. This will result in the system 'hanging' and the only way out of the situation will be to re-boot the system.

Performance testing/load testing/stress testing tools

> **performance testing tool:** *A tool to support performance testing and that usually has two main facilities: load generation and test transaction measurement. Load generation can simulate either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions and these are logged. Performance testing tools normally provide reports based on test logs and graphs of load against response times.*

These tools generate a series of performance measurements (i.e. response times) for a system.

Many of these tools are also used to perform Load and Stress testing as well. For that reason they are often named after the aspect of performance that it measures, such as load or stress, so are also known as load testing tools or stress testing tools. They are often based on automated repetitive execution of tests, controlled by parameters. In order to perform a load or a stress test they must be capable of simulating a large number of users all accessing a system all at once.

If one of the testing requirements that you have is to measure the system's performance, then a performance testing tool is a necessity. Performance testing tools are able to provide extremely accurate measurements of items like response times, service times etc.

There are other tools in this category that are able to simulate large loads including multiple users, heavy network traffic and database accesses. Although these tools are not always that easy to set up, simulating particular loads is usually more cost effective than using a lot of people and/or hardware.

Monitoring tools

Monitoring tools are not strictly testing tools but provide information that can be used for testing purposes and which is not available by other means.

Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems. They store information about the version and build of the software and testware, and enable traceability.

6.1.8. Tool support for specific testing needs

Individual examples of the types of tool classified above can be specialized for use in a particular type of application. For example, there are performance testing tools specifically for web-based applications, static analysis tools for specific development platforms, and dynamic analysis tools specifically for testing security aspects.

Commercial tool suites may target specific application areas (e.g. embedded systems).

6.1.9. Tool support using other tools

The test tools listed so far are not the only types of tools used by testers – they may also use spreadsheets, SQL, resource or debugging tools, for example.

Test Oracle

test oracle: *A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), a user manual, or an individual's specialized knowledge, but should not be the code.*

A test oracle is an (idealised) information source, which delivers the correct reaction (expected results) on input of a test case. In broader terms it is the principle or mechanism by which you recognize a problem.

The ideal solution would be a totally computer supported test oracle. In practice this must be carried out by a person whose knowledge is extracted, from the requirements definition or another source e.g. specification of the test object, prototypes, test generators and the system to be tested itself.

Oracle Heuristics or Possible Oracles

Note. The rest of subsection on test oracles is modified from Black Box Software Testing (Course Notes, Academic Version, 2004) www.testingeducation.org, Copyright (c) Cem Kaner and James Bach, kaner@kaner.com.

The *Oracle assumption* says that the tester is able to routinely identify the correct outcome of a test. The tester can do this by applying heuristics.

A heuristic is a fallible idea or method that may help solve a problem. You don't comply with a heuristic; you apply it. Heuristics are not authoritative rules and should be used carefully. Use heuristics as guidelines when applying them in a specific context.

- Consistent with History: Present function behaviour is consistent with past behaviour.
- Consistent with our Image: Function behaviour is consistent with an image that the organization wants to project.
- Consistent with Comparable Products: Function behaviour is consistent with that of similar functions in comparable products.
- Consistent with Claims: Function behaviour is consistent with documented or advertised behaviour.
- Consistent with Specifications or Regulations: Function behaviour is consistent with statements of what must be met.
- Consistent with User's Expectations: Function behaviour is consistent with what we think users want.
- Consistent within Product: Function behaviour is consistent with behaviour of comparable functions or functional patterns within the product.
- Consistent with Purpose: Function behaviour is consistent with apparent purpose.

Possible Pitfalls

The most common way to look at an oracle is as a source of expected results. After a test has been executed, the results of the test are matched against the ones obtained from the oracle. If they match, the test passes. If they don't match, the test fails. Nevertheless we should take into account that this evaluation is heuristic:

- We can have false alarms: A mismatch between the actual results and the oracle predicted results might not matter.
- We can miss bugs: A match between the results from Test A and Test B might stem from the same error in both tests.

6.2. Effective use of tools: potential benefits and risks

In this session we examine the processes involved in identifying where tools can help the testing process, and how to approach the selection and implementation of those tools.

6.2.1. Potential benefits and risks of tool support for testing (for all tools)

The introduction of test tools can involve large investments therefore the choice of the tools should be carefully and thoughtfully carried out.

Pre-requisite for the successful introduction of test tools is an appropriate maturity level of the test process. A (long-term) cost benefit analysis should be used to judge the profitability of tool usage. Decision criteria for test tools can also be the influence of test quality or strategic factors (e.g. time to market).

Before Buying a Tool

As we have seen there are tools that can help in most areas of testing.

Given that there will be a limit to the budget currently available how do you identify where bringing tools in can help most?

Often the temptation is to automate the execution of tests first, however research has indicated that this, if not thoroughly planned and prepared can lead to money being wasted.

Surveys carried out indicate that while many organisations have Automated Testing tools, many feel that they are not realising the full benefit of those tools, and in a vast number of instances the tools are not used at all!

The selection and implementation of tools is a major task. It is necessary to treat this as a project in its' own right. A number of tools should be evaluated, the best match chosen and then a full implementation plan should be created. Once the tool has been selected it is not a simple case of allowing the users to install it themselves. The tool implementation must be planned and controlled, funding and resources must be made available.

Examine the Test Process

Before testing tools can be successfully implemented it is necessary to ensure that there is a sound testing process in place (or planned for introduction simultaneously with the tool).

Without a sound test process the tool will at best be useless (and swiftly consigned to the shelf). At worse, the tool will accentuate the test process's problems.

You may already have tools in place in certain areas any new tools have to fit in with the existing tools i.e. you may need to integrate the test management tool with a test execution tool with a defect management tool.

Whilst it is theoretically possible to automate almost all of the testing process, it is not normal to do so all at once. Testers will normally select one or two areas that will be the first to gain from automation. These are typically the areas of greatest importance or those that, on a repeated basis, consume the greatest amount of the testers' time.

Examine the Test Environment

Test Environments tend to be one of the most crucial areas for tool implementation – not just the procedures and processes. Shared environments will not accept automation unless the SUT allows the data to be entered fresh every time – this is because the other projects using the environment will not allow back-ups / restores to be done. Automation may also fill an environment up and make it run out of space as it allows a large amount of data to be entered in a short space of time.

As part of the tool evaluation and implementation process it is necessary to review the state of the testing environment(s) are to ensure that it / they are capable of working with testing tools.

Potential benefits of using tools include:

- Repetitive work is reduced (e.g. running regression tests, re-entering the same test data, and checking against coding standards).
- Greater consistency and repeatability (e.g. tests executed by a tool, and tests derived from requirements).
- Objective assessment (e.g. static measures, coverage and system behaviour).

- Ease of access to information about tests or testing (e.g. statistics and graphs about test progress, incident rates and performance).

Risks of using tools include:

- Unrealistic expectations for the tool (including functionality and ease of use).
- Underestimating the time, cost and effort for the initial introduction of a tool (including training and external expertise).
- Underestimating the time and effort needed to achieve significant and continuing benefits from the tool (including the need for changes in the testing process and continuous improvement of the way the tool is used).
- Underestimating the effort required to maintain the test assets generated by the tool.
- Over-reliance on the tool (replacement for test design or where manual testing would be better).

6.2.2. Special considerations for some types of tool

Test execution tools

Test execution tools replay scripts designed to implement tests that are stored electronically. This type of tool often requires significant effort in order to achieve significant benefits.

Capturing tests by recording the actions of a manual tester seems attractive, but this approach does not scale to large numbers of automated tests. A captured script is a linear representation with specific data and actions as part of each script. This type of script may be unstable when unexpected events occur.

A data-driven approach separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic script that can read the test data and perform the same test with different data. Testers who are not familiar with the scripting language can enter test data for these predefined scripts.

In a keyword-driven approach, the spreadsheet contains keywords describing the actions to be taken (also called action words), and test data. Testers (even if they are not familiar with the scripting language) can then define tests using the keywords, which can be tailored to the application being tested.

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

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

Performance testing tools

Performance testing tools need someone with expertise in performance testing to help design the tests and interpret the results.

Static analysis tools

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

Test management tools

Test management tools need to interface with other tools or spreadsheets in order to produce information in the best format for the current needs of the organization. The reports need to be designed and monitored so that they provide benefit.

6.3. Introducing a tool into an organization

Before introducing a tool into an organization the right tool should be selected from all available on the market.

Tool selection should be executed in six steps:

1. Begin with a **requirements specification** detailing the use of the tool. Build upon the knowledge gathered during the initial analyses described above.
2. Conduct a **market analysis** of the currently available tools.
3. Arrange **demonstrations** of the favourite tools with the manufacturer.
4. Subsequently conduct **evaluation** of a smaller selection of tools.
5. A **review** follows.
6. Assuming a suitable tool is found, the **final tool selection** is made.

Market Analysis and Funding

Software testing tools can typically cost from $ 1,000 (for a single PC seat) to in excess of $ 500,000. Therefore before demonstrations and trials commence it is necessary to ensure that the commitment for funding is there. (Most vendors in the PC market are often willing to provide evaluation copies in addition to doing demos.)

In addition to the cost of the tool, it is necessary to budget for the resources that are necessary to implement the tool and actually start using it:

- There may be some consultancy charges for implementation.
- In order to get maximum benefit and reduce the learning curve training is a must.
- The company will need to allow for some resources to actually start using the tool.

Demonstrations Evaluations and Review

These three steps are closely linked together. Normally you will identify a number of potential products. You will have a subset of them demonstrated to you. Then you will evaluate a few of the tools and will end up with a couple of favourites. During the review these two or three tools will be compared by several stakeholders in your organizations.

The end result of the review should be a recommendation to select one of the tools (or to continue searching if none of the tools seems acceptable.)

After selecting the most appropriate tool it has to be introduced into the organization.
The main principles are:

- Assessment of organizational maturity, strengths and weaknesses and identification of opportunities for an improved test process supported by tools.

- Evaluation against clear requirements and objective criteria.
- A proof-of-concept to test the required functionality and determine whether the product meets its objectives.
- Evaluation of the vendor (including training, support and commercial aspects).
- Identification of internal requirements for coaching and mentoring in the use of the tool.

The proof-of-concept could be done in a small-scale **pilot project**, making it possible to minimize impacts if major hurdles are found and the pilot is not successful.

A pilot project has the following objectives:

- Learn more detail about the tool.
- See how the tool would fit with existing processes and practices, and how they would need to change.
- Decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g. deciding on naming conventions for files and tests, creating libraries and defining the modularity of test suites).
- Assess whether the benefits will be achieved at reasonable cost.

Success factors for the deployment of the tool within an organization include:

- Rolling out the tool to the rest of the organization incrementally.
- Adapting and improving processes to fit with the use of the tool.
- Providing training and coaching/mentoring for new users.
- Defining usage guidelines.
- Implementing a way to learn lessons from tool use.
- Monitoring tool use and benefits.

References

6.2.2. – Buwalda, 2001, Fewster, 1999

6.3. - Fewster, 1999