Plan and design software testing activities
Unit Standard 386056

The purpose of the unit standard:

- Explaining the principles of software testing.
- Performing software testing practices.
- Implementing software testing levels.
- Generating a system test plan.
- Explaining test case design techniques.
- Preparing a systems test design

Explain the principles of software testing
Specific Outcome 1

Learning Outcomes

- Software testing is defined in terms of its purpose and objectives.
- Software test requirements and conditions are explained according to the functionality of the software.
- Software test scripts are defined according to international conventions.
- Software test scripts and procedures are explained with examples.
- The use of testware is explained in terms of relationships with other components.
- The relationship between testware and software testing process is explored using test requirements.

Software Testing

Software testing is a process of executing a program or application with the intent of finding the software bugs.

- It can also be stated as the process of validating and verifying that a software program or application or product:
- Meets the business and technical requirements that guided it's design and development
- Works as expected
- Can be implemented with the same characteristic.

Let's break the definition of Software testing into the following parts:

1) **Process**: Testing is a process rather than a single activity.

2) **All Life Cycle Activities**: Testing is a process that's take place throughout the Software Development Life Cycle (SDLC).

The process of designing tests early in the life cycle can help to prevent defects from being introduced in the code. Sometimes it's referred as "verifying the test basis via the test design".
The test basis includes documents such as the requirements and design specifications.
Software Testing 3) Static **Testing**: It can test and find defects without executing code. Static Testing is done during verification process. This testing includes reviewing of the documents (including source code) and static analysis. This is useful and cost-effective way of testing. For example: reviewing, walkthrough, inspection, etc.

4) **Dynamic Testing**: Testing that involves the execution of the test item. In dynamic testing the software code is executed to demonstrate the result of running tests. It's done during validation process. For example: unit testing, integration testing, system testing, etc.

5) **Planning:** We need to plan as what we want to do. We control the test activities; we report on testing progress and the status of the software under test.

6) **Preparation:** We need to choose what testing we will do, by selecting test conditions and designing test cases.

7) **Evaluation:** During evaluation we must check the results and evaluate the software under test and the completion criteria, which helps us to decide whether we have finished testing and whether the software product has passed the tests.

8) **Software products and related work products**: Along with the testing of code the testing of requirement and design specifications and also the related documents like operation, user and training material is equally important.

the principles of testing

Principles of Testing – There are seven principles of testing. They are as follows:

1) **Testing shows presence of defects**: Testing can show the defects are present but cannot prove that there are no defects. Even after testing the application or product thoroughly we cannot say that the product is 100% defect free. Testing always reduces the number of undiscovered defects remaining in the software but even if no defects are found, it is not a proof of correctness.

2) **Exhaustive testing is impossible**: Testing everything including all combinations of inputs and preconditions is not possible. So, instead of doing the exhaustive testing we can use risks and priorities to focus testing efforts. For example: In an application in one screen there are 15 input fields, each having 5 possible values, then to test all the valid combinations you would need 30 517 578 125 (515) tests. This is very unlikely that the project timescales would allow for this number of tests. So, accessing and managing risk is one of the most important activities and reason for testing in any project.

3) **Early testing**: In the software development life cycle testing activities should start as early as possible and should be focused on defined objectives.

4) **Defect clustering**: A small number of modules contains most of the defects discovered during pre-release testing or shows the most operational failures.

5) **Pesticide paradox:** If the same kinds of tests are repeated again and again, eventually the same set of test cases will no longer be able to find any new bugs. To overcome this “Pesticide Paradox”, it is really very important to review the test cases regularly and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.

6) **Testing is context dependent**: Testing is basically context dependent. Different kinds of sites are tested differently. For example, safety – critical software is tested differently from an e-commerce site.

7) **Absence – of – errors fallacy**: If the system built is unusable and does not fulfil the user's needs and expectations then finding and fixing defects does not help.

Other principles to note are:

Testing must be done by an independent party.

Testing should not be performed by the person or team that developed the software since they tend to defend the correctness of the program.

Assign best personnel to the task.

Because testing requires high creativity and responsibility only the best personnel must be assigned to design, implement, and analyse test cases, test data and test results.

Test for invalid and unexpected input conditions as well as valid conditions.

The program should generate correct messages when an invalid test is encountered and should generate correct results when the test is valid.

Keep software static during test.

The program must not be modified during the implementation of the set of designed test cases.

Provide expected test results if possible.

A necessary part of test documentation is the specification of expected results, even if providing such results is impractical. The program should generate correct messages when an invalid test is encountered and should generate correct results when the test is valid.

Keep software static during test.

The program must not be modified during the implementation of the set of designed test cases.

Provide expected test results if possible.

A necessary part of test documentation is the specification of expected results, even if providing such results is impractical.

Requirements Testing

Requirements testing is done to clarify whether project requirements are feasible or not in terms of time, resources and budget.

Many bugs emerge in software because of incompleteness, inaccuracy and ambiguities in functional requirements.

That's why it is highly important to test requirements and eliminate ambiguities before you start to develop a project.

This type of testing covers testing of requirements specification that describes:

- project functionality
- user interface
- software and hardware interfaces
- performance criteria
- implementation issues and risks
- security and system correctness criteria

Requirements testing also includes help in gathering and analysis of user data and their domain. This procedure is required to research UI usability, creation of profiles, finalizing documentation and, on the whole, facilitates design process.

This procedure helps improve requirements quality and reduce the number of tests necessary to meet these requirements.

First of all, our QA team gathers and analyses requirements to find out what technology will be used to build the project, who this project is designed for and the project goal.

We also discover missing requirements to make sure the information we gathered is consistent, clear and covers every aspect of software under development.

Then we create test cases on the basis of the requirement measurements and perform them to make sure that the system under test operates in accordance with the set requirements.

For example, you'd like to have system response time of 2 seconds – that is your requirement measurement. All of such requirement measurements are passed through tests and then evaluated.

We also take part in review meetings, and in case any new requirements appear, or any changes are made to the set requirements, we track them in an issue tracking system.

Finally, when all requirements are tested and the client is satisfied with the results and signs the requirements document, we freeze the requirements for design phase.

Test script in terms of software testing

A test script is a set of instructions, written using a scripting or programming language, that are executed on a system under test. A test script is used to verify that the system performs as expected. Now a days, QA outsourcing companies are primarily focusing on automation. Some of the most commonly used scripting languages used in automation testing are:

- Perl
- Java
- Ruby
- Python
- VB script

There are many automation tools that generate the test scripts in their own scripting languages without the need for actual coding, for example Selenium IDE.

Testware

Generally speaking, Testware is a work products produced during the test process for use in planning, designing, executing, evaluating and reporting on testing, especially for software testing automation. Automation testware for example is designed to be executed on automation frameworks. Testware is an umbrella term for all utilities and application software that serve in combination for testing a software package but not necessarily contribute to operational purposes. As such, testware is not a standing configuration but merely a working environment for application software or subsets thereof.

It includes artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.

Testware is produced by both verification and validation testing methods. Like software, Testware includes codes and binaries as well as test cases, test plan, test report, etc. Testware should be placed under the control of a configuration management system, saved and faithfully maintained.

Compared to general software, testware is special because it has:

- a different purpose
- different metrics for quality and
- different users

The different methods should be adopted when you develop testware with what you use to develop general software.

Testware is also referred as test tools in a narrow sense.

Generate a system test plan
Specific Outcome 2

Learning Outcomes

- A software test strategy is developed in line with test requirements and specifications.
- Test documentation is identified and developed according to test requirements.
- A systems test plan is created to meet test planning and design requirements.

Recording test execution results is very important part of testing, whenever test execution cycle is complete, tester should make a complete test results report which includes the Test Pass/Fail status of the test cycle.

If manual testing is done, then the test pass/fail result should be captured in an excel sheet and if automation testing is done using automation tool then the HTML or XML reports should be provided to stakeholders as test deliverable.

Test results

Testing results include the following:

- Test plan status
- Test documentation status
- Test execution status (defect status)

Test Plan: It is enough to communicate with the rest of the project teams, when a test plan is created or when a major change is made to it.

Test documentation – Let all the teams know when the designing of the tests, data gathering, and other activities have begun and also when they are finished. This report will not only let them know about the progress of the task but also signal the teams that need to review and provide signoff on the artefacts, that they are up next.

Test execution– Execution is the phase of a project when the testing team is the primary **focus** – positively and negatively – we are both the heroes and the villains.

A typical day during a test cycle is not done, unless the daily status report is sent out. In some teams, they could agree on a weekly report, but having it sent daily is the norm.

What should programme test result show?

a) Number of tests done

b) Number of test cases executed

c) Number of defects encountered that day/and their respective states

d) Number of defect encountered so far/and their respective states

e) Number of critical defects- still open

f) Environment downtimes – if any

g) Showstoppers – if any

h) Attachment of the test execution sheet / Link to the test management tool where the test cases are placed

i) Attachment to the bug report/link to the defect/test management tool used for incident management

Few pointers to help the process along:

a) Be concise at the same time complete

b) Make sure the results you report are accurate

c) Use bulleted points to make the report very readable

d) Double check to include the right date, subject, to list and attachments.

e) If the report is too big and has too many factors to report: place it in a common location as a file and send a link in the email instead of the file itself. (Be sure the recipients have access permissions to this location and the file)

f) If it is a status meeting – Be prepared for the presentation, arrive on time and most importantly, maintain an even tone (don't be too proud of the defects – they are in general "bad news").

Below are samples reports of programme test results.

Module	Scenarios	Sub Levels	Complexity	Responsible tester	Date of Execution(Can be past, present or future date)	Status(Pass/Fail/blocked/not executed)	Defect ID- Brief description	Severity	Status
Admin	Login Page	Login	Medium	Tester A	31-08-2013	Pass			
Admin	Country Management	Add country	Complex	Tester B	31-08-2013	Pass			
		Delete country	Complex	Tester B	31-08-2013	Pass			
		Verify the list display	Medium	Tester B	31-08-2013	Pass			
Admin	City management	Add City	Complex	Tester A	31-08-2013	Pass			
		Delete City	Complex	Tester A	31-08-2013	Pass			
		Verify the list display	Medium	Tester A	31-08-2013	Pass			
Admin	Car Make Management	Add Car Make	Complex	Tester B	31-08-2013	Fail	1028: Cannot add car make, 404 error on clicking the link	1- High	Open
		Delete car make	Complex	Tester B	31-08-2013	Blocked	1028		
		Verify the list display	Medium	Tester B	31-08-2013	Pass			

SoftwareTestingHelp.com

Test ID _____ Test System ID: _____

Test Objective: _____

Specification: _____

Step	Test Procedure	Expected Result	Actual Result	Required documents	Pass/fail
1					
2					

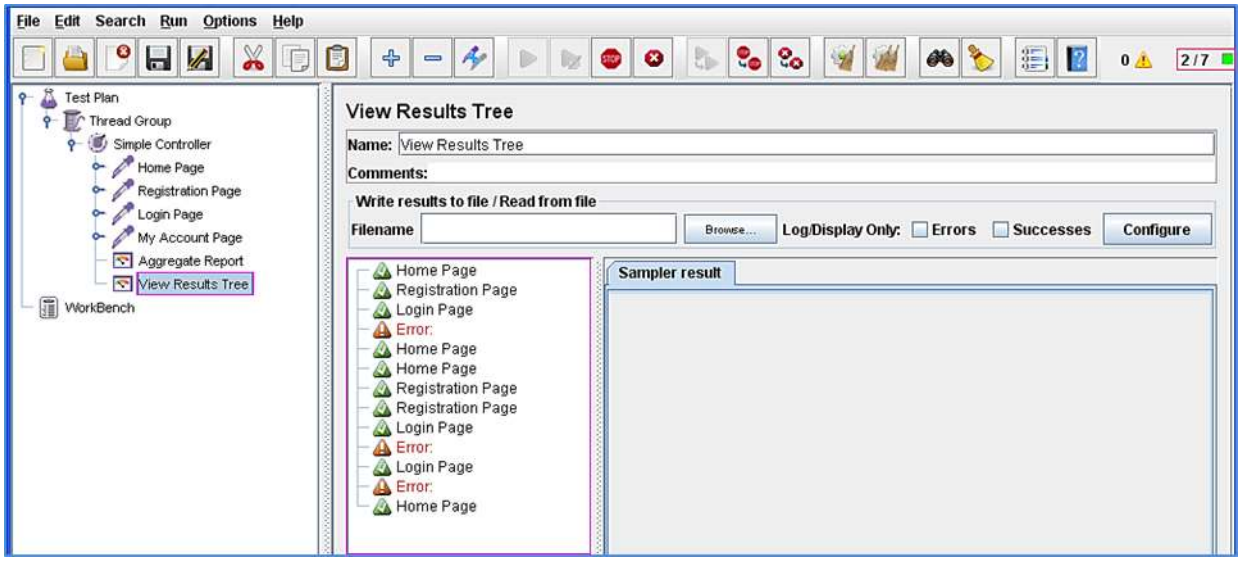
Tester: I confirm that I have all tests executed as described

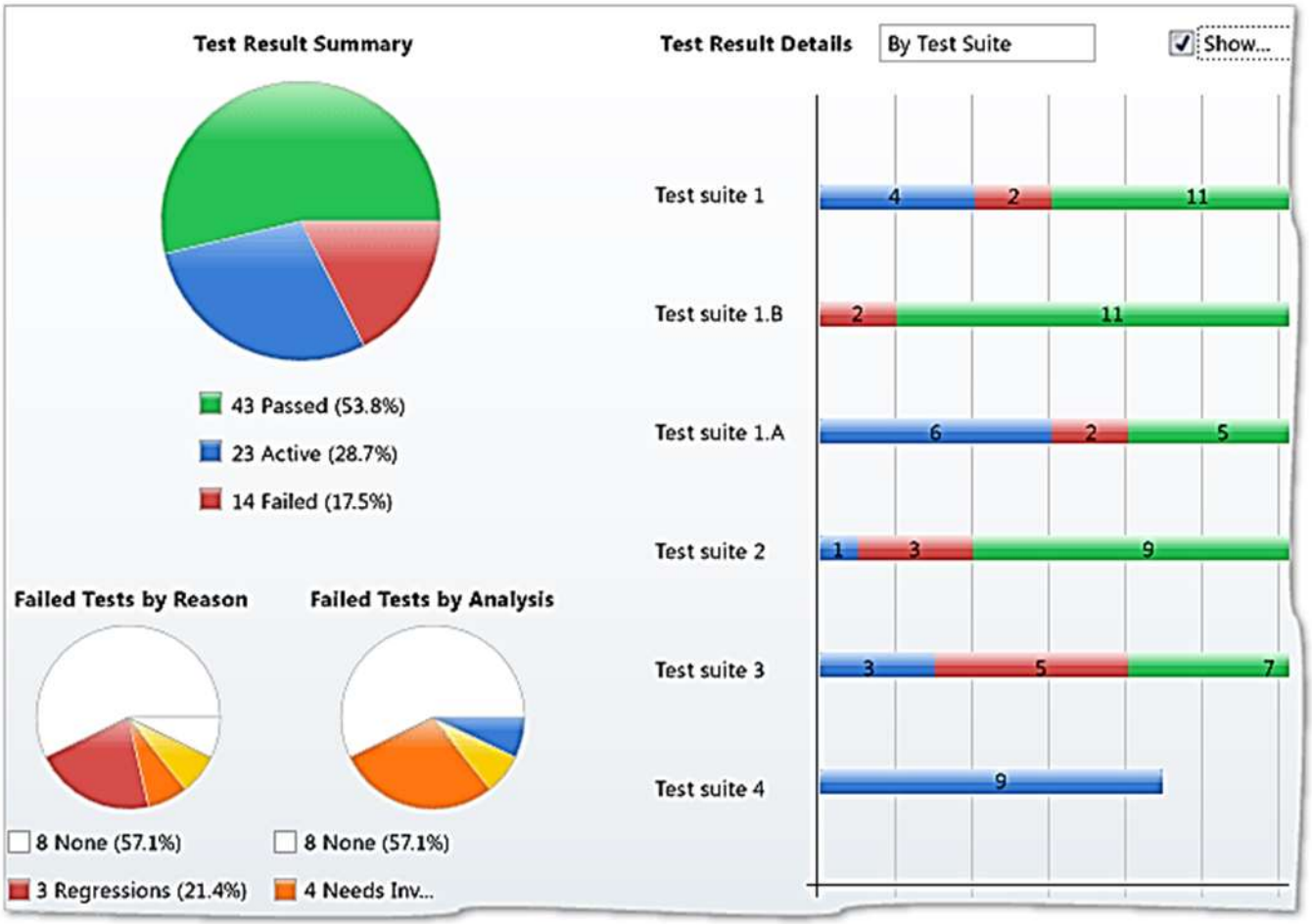
Name: _____ Signature_____ Date_____

Tests passed: yes no Comment: _____

Reviewer: I confirm that I have reviewed test documentation

Name: _____ Signature Date




Test Result Summary
Test Result Details
By Test Suite
Show...
43 Passed (53.8%)
23 Active (28.7%)
14 Failed (17.5%)
Test suite 1
4
2
11
Test suite 1.B
2
11
Test suite 1.A
6
2
5
Test suite 2
1
3
9
Test suite 3
3
5
7
Test suite 4
9
Failed Tests by Reason
Failed Tests by Analysis
8 None (57.1%)
3 Regressions (21.4%)
8 None (57.1%)
4 Needs Inv...

Explain test case design techniques
Specific Outcome 3

Learning Outcomes

- Strategies for generating test cases are explained with examples.
- Different types of tests are described with an example of each.
- Test case design techniques are executed in accordance with test requirements.

Programme Testing refers to a set of activities conducted with the intent of finding errors in software.

Test plan refers to specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Static vs. dynamic testing

There are many approaches available in software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing is often implicit, as proofreading, plus when programming tools/text editors check source code structure or compilers (pre-compilers) check syntax and data flow as static program analysis. Dynamic testing takes place when the program itself is run. Dynamic testing may begin before the program is 100% complete in order to test particular sections of code and are applied to discrete functions or modules. Typical techniques for this are either using stubs/drivers or execution from a debugger environment.

Static testing involves verification, whereas dynamic testing involves validation. Together they help improve software quality. Among the techniques for static analysis, mutation testing can be used to ensure the test-cases will detect errors which are introduced by mutating the source code.

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White-Box testing

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing** and **structural testing**) Testing based on an analysis of the internal structure of the component or system, as opposed to the functionality exposed to the end-user. In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test.

Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

Techniques used in white-box testing include:

- API testing (application programming interface) – testing of the application using public and private APIs
- Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
- Mutation testing methods
- Static testing methods
- Code coverage tools can evaluate the completeness of a test suite that was created with any method, including black-box testing. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested. Code coverage as a software metric can be reported as a percentage for:
  - *Function coverage*, which reports on functions executed
  - *Statement coverage*, which reports on the number of lines executed to complete the test.
  - 100% statement coverage ensures that all code paths or branches (in terms of control flow) are executed at least once. This is helpful in ensuring correct functionality, but not sufficient since the same code may process different inputs correctly or incorrectly.

Black-box testing

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The testers are only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: partitioning, boundary, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory and specification-based testing.

Specification-based testing

Aims to test the functionality of software according to the applicable requirements. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behaviour), either "is" or "is not" the same as the expected value specified in the test case. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional, though usually functional.

Specification-based testing may be necessary to assure correct functionality, but it is insufficient to guard against complex or high-risk situations.

One advantage of the black box technique is that no programming knowledge is required. Whatever biases the programmers may have had, the tester likely has a different set and may emphasize different areas of functionality. On the other hand, black-box testing has been said to be "like a walk in a dark labyrinth without a flashlight. Because they do not examine the source code, there are situations when a tester writes many test cases to check something that could have been tested by only one test case, or leaves some parts of the program untested.

This method of test can be applied to all levels of software
testing: unit, integration, system and acceptance. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

Visual testing

The aim of visual testing is to provide developers with the ability to examine what was happening at the point of software failure by presenting the data in such a way that the developer can easily find the information he or she requires, and the information is expressed clearly.

At the core of visual testing is the idea that showing someone a problem (or a test failure), rather than just describing it, greatly increases clarity and understanding. Visual testing therefore requires the recording of the entire test process – capturing everything that occurs on the test system in video format. Output videos are supplemented by real-time tester input via picture-in-a-picture webcam and audio commentary from microphones.

Visual testing provides a number of advantages. The quality of communication is increased dramatically because testers can show the problem (and the events leading up to it) to the developer as opposed to just describing it and the need to replicate test failures will cease to exist in many cases.

The developer will have all the evidence he or she requires of a test failure and can instead focus on the cause of the fault and how it should be fixed.

Visual testing is particularly well-suited for environments that deploy agile methods in their development of software, since agile methods require greater communication between testers and developers and collaboration within small teams.

Ad hoc testing and exploratory testing are important methodologies for checking software integrity, because they require less preparation time to implement, while the important bugs can be found quickly. In ad hoc testing, where testing takes place in an improvised, impromptu way, the ability of a test tool to visually record everything that occurs on a system becomes very important.

Visual testing is gathering recognition in customer acceptance and usability testing, because the test can be used by many individuals involved in the development process. For the customer, it becomes easy to provide detailed bug reports and feedback, and for program users, visual testing can record user actions on screen, as well as their voice and image, to provide a complete picture at the time of software failure for the developer.

Grey-box testing

Grey-box testing involves having knowledge of internal data structures and algorithms for purposes of designing tests, while executing those tests at the user, or black-box level. The tester is not required to have full access to the software's source code. Manipulating input data and formatting output do not qualify as grey-box, because the input and output are clearly outside of the "black box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test.

However, tests that require modifying a back-end data repository such as a database or a log file does qualify as grey-box, as the user would not normally be able to change the data repository in normal production operations Grey-box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

By knowing the underlying concepts of how the software works, the tester makes better-informed testing choices while testing the software from outside. Typically, a grey-box tester will be permitted to set up an isolated testing environment with activities such as seeding a database. The tester can observe the state of the product being tested after performing certain actions such as executing SQL statements against the database and then executing queries to ensure that the

expected changes have been reflected. Grey-box testing implements intelligent test scenarios, based on limited information. This will particularly apply to data type handling, exception handling, and so on.

Component interface testing

The practice of component interface testing can be used to check the handling of data passed between various units, or subsystem components, beyond full integration testing between those units. The data being passed can be considered as "message packets" and the range or data types can be checked, for data generated from one unit, and tested for validity before being passed into another unit.

One option for interface testing is to keep a separate log file of data items being passed, often with a timestamp logged to allow analysis of thousands of cases of data passed between units for days or weeks. Tests can include checking the handling of some extreme data values while other interface variables are passed as normal values. Unusual data values in an interface can help explain unexpected performance in the next unit. Component interface testing is a variation of **black-box testing** with the focus on the data values beyond just the related actions of a subsystem component.

Installation testing

An installation test assures that the system is installed correctly and working at actual customer's hardware.

Compatibility testing

A common cause of software failure (real or perceived) is a lack of its compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser).

For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactive abstracting operating system functionality into a separate program module or library.

Smoke and sanity testing

Sanity testing determines whether it is reasonable to proceed with further testing.

Smoke testing consists of minimal attempts to operate the software, designed to determine whether there are any basic problems that will prevent it from working at all. Such tests can be used as build verification test.

Regression testing
A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software. Specifically, it seeks to uncover software regressions, as degraded or lost features, including old bugs that have come back. Such regressions occur whenever software functionality that was previously working, correctly, stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previous sets of test-cases and checking whether previously fixed faults have re-emerged.

The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, or be very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Regression testing is typically the largest test effort in commercial software development due to checking numerous details in prior software features, and even new software can be developed while using some old test-cases to test parts of the new design to ensure prior functionality is still supported.

Alpha testing
A type of acceptance testing performed in the developer's test environment by roles outside the development organization.

Beta testing
A type of acceptance testing performed at an external site to the developer's test environment by roles outside the development organization. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Functional vs non-functional testing
Functional testing refers to a testing performed to evaluate if a component or system satisfies functional requirements. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work."

Non-functional testing refers to a testing performed to evaluate that a component or system complies with non-functional requirements, such as scalability or other performance, behaviour under certain constraints, or security. Testing will determine the breaking point, the point at which extremes of

scalability or performance leads to unstable execution. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail. It verifies that the software functions properly even when it receives invalid or unexpected inputs, thereby establishing the robustness of input validation and error-management routines. Software fault injection, in the form of fuzzing, is an example of failure testing. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform destructive testing.

Software performance testing

Testing to determine the performance efficiency of a component or system. Performance testing is generally executed to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage.

Load testing is a type of performance testing conducted to evaluate the behavior of a component or system under varying loads, usually between anticipated conditions of low, typical, and peak usage. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*. Volume is a way to test software functions even when certain components (for example a file or database) increase radically in size. *Stress testing* is a way to test reliability under unexpected or rare workloads. *Stability testing* (often referred to as load or endurance testing) checks to see if the software can continuously function well in or above an acceptable period.

There is little agreement on what the specific goals of performance testing are. The terms load testing, performance testing, scalability testing, and volume testing, are often used interchangeably.
Real-time software systems have strict timing constraints. To test if timing constraints are met, real-time testing is used.

Usability testing

Testing to evaluate the degree to which the system can be used by specified users with effectiveness, efficiency and satisfaction in a specified context of use.

Security testing

Testing to determine the security of the software product. Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).

Actual translation to human languages must be tested, too. Possible localization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- Technical terminology may become inconsistent if the project is translated by several people without proper coordination or if the translator is imprudent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically at run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut which has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes which are appropriate in the source language may be inappropriate in the target language; for example, CJK characters may become unreadable if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to crash or malfunction.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that was not localized.
- Localized operating systems may have differently named system configuration files and environment variables and different formats for date and currency.

Important terms and concepts in programme testing

Acceptance Testing: A test level that focuses on determining whether to accept the system. Normally performed to validate the software meets a set of agreed acceptance criteria.

Accessibility Testing: Testing to determine the ease by which users with disabilities can use a component or system.

Ad Hoc Testing: A testing phase where the tester tries to 'break' the system by randomly trying the system's functionality. Can include negative testing as well. See also Monkey Testing.

Agile Testing: Testing practice for projects using agile methodologies, treating development as the customer of testing and emphasizing a test-first design paradigm. See also Test Driven Development.

Application Binary Interface (ABI): A specification defining requirements for portability of applications in binary forms across different system platforms and environments.

Application Programming Interface (API): A formalized set of software calls and routines that can be referenced by an application program in order to access supporting system or network services.

Automated Software Quality (ASQ): The use of software tools, such as automated testing tools, to improve software quality.

Automated Testing:

Testing employing software tools which execute tests without manual intervention. Can be applied in GUI, performance, API, etc. testing.

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.

Backus-Naur Form: A metalanguage used to formally describe the syntax of a language.

Basic Block: A sequence of one or more consecutive, executable statements containing no branches.

Basis Path Testing: A white box test case design technique that uses the algorithmic flow of the program to design tests.

Basis Set: The set of tests derived using basis path testing.

Baseline: The point at which some deliverable produced during the software engineering process is put under formal change control.

Benchmark Testing: Tests that use representative sets of programs and data designed to evaluate the performance of computer hardware and software in a given configuration.

Beta Testing: A type of acceptance testing performed at an external site to the developer's test environment by roles outside the development organization.

Binary Portability Testing: Testing an executable application for portability across system platforms and environments, usually for conformation to an ABI specification.

Black Box Testing: Testing based on an analysis of the specification of a piece of software without reference to its internal workings. The goal is to test how well the component conforms to the published requirements for the component.

Bottom Up Testing: An approach to integration testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

Boundary Testing: Test which focus on the boundary or limit conditions of the software being tested. (Some of these tests are stress tests).

Boundary Value Analysis: A black-box test technique in which test cases are designed based on boundary values.

Branch Testing: Testing in which all branches in the program source code are tested at least once.

Breadth Testing: A test suite that exercises the full functionality of a product but does not test features in detail.

Bug: A fault in a program which causes the program to perform in an unintended or unanticipated manner.

CAST: Computer Aided Software Testing.

Capture/Replay Tool: A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. Most commonly applied to GUI test tools.

CMM: The Capability Maturity Model for Software (CMM or SW-CMM) is a model for judging the maturity of the software processes of an organization and for identifying the key practices that are required to increase the maturity of these processes.

Cause Effect Graph: A graphical representation of inputs and the associated outputs effects which can be used to design test cases.

Code Complete: Phase of development where functionality is implemented in entirety; bug fixes are all that are left. All functions found in the Functional Specifications have been implemented.

Code Coverage: The coverage of code. An analysis method that determines which parts of the software have been executed (covered) by the test case suite and which parts have not been executed and therefore may require additional attention.

Code Inspection: A formal testing technique where the programmer reviews source code with a group who ask questions analysing the program logic, analysing the code with respect to a checklist of historically common programming errors, and analysing its compliance with coding standards.

Code Walkthrough: A formal testing technique where source code is traced by a group with a small set of test cases, while the state of program variables is manually monitored, to analyse the programmer's logic and assumptions.

Coding: The generation of source code.

Compatibility Testing: Testing whether software is compatible with other elements of a system with which it should operate, e.g. browsers, Operating Systems, or hardware.

Component: A part of a system that can be tested in isolation.

Component Testing: A test level that focuses on individual hardware or software components.

Concurrency Testing: Multi-user testing geared towards determining the effects of accessing the same application code, module or database records. Identifies and measures the level of locking, deadlocking and use of single-threaded code and locking semaphores.

Conformance Testing: The process of testing that an implementation conforms to the specification on which it is based. Usually applied to testing conformance to a formal standard.

Context Driven Testing: The context-driven school of software testing is flavour of Agile Testing that advocates continuous and creative evaluation of testing opportunities in light of the potential information revealed and the value of that information to the organization right now.

Conversion Testing: Testing of programs or procedures used to convert data from existing systems for use in replacement systems.

Cyclomatic Complexity: A measure of the logical complexity of an algorithm, used in white-box testing.

Data Dictionary: A database that contains definitions of all data items defined during analysis.

Data Flow Diagram: A modelling notation that represents a functional decomposition of a system.

Data Driven Testing: A scripting technique that uses data files to contain the test data and expected results needed to execute the test scripts.

Debugging: The process of finding, analyzing and removing the causes of failures in a component or system.

Defect: Non-conformance to requirements or functional / program specification

Dependency Testing: Examines an application's requirements for pre-existing software, initial states and configuration in order to maintain proper functionality.

Depth Testing: A test that exercises a feature of a product in full detail.

Dynamic Testing: Testing that involves the execution of the test item.

Emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Endurance Testing: Checks for memory leaks or other problems that may occur with prolonged execution.

End-to-End testing: Testing a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Equivalence Class: A portion of a component's input or output domains for which the component's behaviour is assumed to be the same from the component's specification.

Equivalence Partitioning: A black-box test technique in which test cases are designed to exercise equivalence partitions by using one representative member of each partition.

Error: A human action that produces an incorrect result.

Exhaustive Testing: A test approach in which the test suite comprises all combinations of input values and preconditions.

Functional Decomposition: A technique used during planning, analysis and design; creates a functional hierarchy for the software.

Functional Specification: A document that describes in detail the characteristics of the product with regard to its intended features.

Functional Testing: See also Black Box Testing.

Testing performed to evaluate if a component or system satisfies functional requirements.

Glass Box Testing: A synonym for White Box Testing.

Gorilla Testing: Testing one particular module, functionality heavily.

Grey Box Testing: A combination of Black Box and White Box testing methodologies: testing a piece of software against its specification but using some knowledge of its internal workings.

High Order Tests: Black-box tests conducted once the software has been integrated.

Independent Test Group (ITG): A group of people whose primary responsibility is software testing.

Inspection: A type of formal review to identify issues in a work product, which provides measurement to improve the review process and the software development process.

Integration Testing: A test level that focuses on interactions between components or systems.

Installation Testing: Confirms that the application under test recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Load Testing: See Performance Testing.

Localization Testing: This term refers to making software specifically designed for a specific locality.

Loop Testing: A white box testing technique that exercises program loops.

Metric: A measurement scale and the method used for measurement.

Monkey Testing: Testing a system or an Application on the fly, i.e just few tests here and there to ensure the system or an application does not crash out.

Mutation Testing: Testing done on the application where bugs are purposely added to it.

Negative Testing: Testing aimed at showing software does not work. Also known as "test to fail". See also Positive Testing.

N+1 Testing: A variation of Regression Testing. Testing conducted with multiple cycles in which errors found in test cycle N are resolved and the solution is retested in test cycle N+1. The cycles are typically repeated until the solution reaches a steady state and there are no errors. See also Regression Testing.

Path Testing: Testing in which all paths in the program source code are tested at least once.

Performance Testing: Testing to determine the performance efficiency of a component or system.

Positive Testing: Testing aimed at showing software works. Also known as "test to pass".

Quality Assurance: Activities focused on providing confidence that quality requirements will be fulfilled.

Quality Audit: A systematic and independent examination to determine whether quality activities and related results comply with planned arrangements and whether these arrangements are implemented effectively and are suitable to achieve objectives.

Quality Circle: A group of individuals with related interests that meet at regular intervals to consider problems or other matters related to the quality of outputs of a process and to the correction of problems or to the improvement of quality.

Quality Control: A set of activities designed to evaluate the quality of a component or system.

Quality Management: Coordinated activities to direct and control an organization with regard to quality that include establishing a quality policy and quality objectives, quality planning, quality control, quality assurance, and quality improvement.

Quality Policy: The overall intentions and direction of an organization as regards quality as formally expressed by top management.

Quality System: The organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

Race Condition: A cause of concurrency problems. Multiple accesses to a shared resource, at least one of which is a write, with no mechanism used by either to moderate simultaneous access.

Ramp Testing: Continuously raising an input signal until the system breaks down.

Recovery Testing: Confirms that the program recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Regression Testing: A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software.

Release Candidate: A pre-release version, which contains the desired functionality of the final version, but which needs to be tested for bugs (which ideally should be removed before the final version is released).

Sanity Testing: Brief test of major functional elements of a piece of software to determine if it's basically operational. See also Smoke Testing.

Scalability Testing: Performance testing focused on ensuring the application under test gracefully handles increases in work load.

Security Testing: Testing to determine the security of the software product.

Smoke Testing: A quick-and-dirty test that the major functions of a piece of software work. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire.

Soak Testing: Running a system at high load for a prolonged period of time. For example, running several times more transactions in an entire day (or night) than would be expected in a busy day,

to identify and performance problems that appear after a large number of transactions have been executed.

Software Requirements Specification: A deliverable that describes all data, functional and behavioural requirements, all constraints, and all validation requirements for software/

Static Analysis: The process of evaluating a component or system without executing it, based on its form, structure, content, or documentation.

Static Analyser: A tool that carries out static analysis.

Static Testing: Testing a work product without the work product code being executed.

Storage Testing: Testing that verifies the program under test stores data files in the correct directories and that it reserves sufficient space to prevent unexpected termination resulting from lack of space. This is external storage as opposed to internal storage.

Stress Testing: Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. Often this is performance testing using a very high level of simulated load.

Structural Testing: Testing based on an analysis of internal workings and structure of a piece of software. See also White Box Testing.

System Testing: A test level that focuses on verifying that a system as a whole meets specified requirements

Testability: The degree to which test conditions can be established for a component or system, and tests can be performed to determine whether those test conditions have been met.

Testing:

The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

Test Bed: An execution environment configured for testing. May consist of specific hardware, OS, network topology, configuration of the product under test, other application or system software, etc. The Test Plan for a project should enumerated the test beds(s) to be used.

Test Case:

A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

Test Driven Development: Testing methodology associated with Agile Programming in which every chunk of code is covered by unit tests, which must all pass all the time, in an effort to eliminate unit-level and regression bugs during development. Practitioners of TDD write a lot of tests, i.e. an equal number of lines of test code to the size of the production code.

Test Driver: A program or test tool used to execute a tests. Also known as a Test Harness.

Test Environment: An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.

Test First Design: Test-first design is one of the mandatory practices of Extreme Programming (XP).It requires that programmers do not write any production code until they have first written a unit test.

Test Harness: A collection of stubs and drivers needed to execute a test suite

Test Plan: Documentation describing the test objectives to be achieved and the means and the schedule for achieving them, organized to coordinate testing activities.

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

Test Scenario: Definition of a set of test cases or test scripts and the sequence in which they are to be executed.

Test Script: A sequence of instructions for the execution of a test

Test Specification: A document specifying the test approach for a software feature or combination or features and the inputs, predicted results and execution conditions for the associated tests.

Test Suite: A set of test scripts or test procedures to be executed in a specific test run.

Test Tools: Computer programs used in the testing of a system, a component of the system, or its documentation.

Thread Testing: A variation of top-down testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by successively lower levels.

Top Down Testing: An 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.

Total Quality Management: A company commitment to develop a process that achieves high quality product and customer satisfaction.

Traceability Matrix: A document showing the relationship between Test Requirements and Test Cases.

Usability Testing: Testing to evaluate the degree to which the system can be used by specified users with effectiveness, efficiency and satisfaction in a specified context of use.

Use Case: A set of possible interactions between an external actor and a system that provides a benefit for the actor involved.

User Acceptance Testing: A type of acceptance testing performed to determine if intended users accept the system.

Unit Testing: Testing of individual software components.

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

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

Volume Testing: Testing which confirms that any values that may become large over time (such as accumulated counts, logs, and data files), can be accommodated by the program and will not cause the program to stop working or degrade its operation in any manner.

Walkthrough: A review of requirements, designs or code characterized by the author of the material under review guiding the progression of the review.

White Box Testing: Testing based on an analysis of the internal structure of the component or system.

Workflow Testing: Scripted end-to-end testing which duplicates specific workflows which are expected to be utilized by the end-user.

Determine software testing levels
Specific Outcome 4

Learning Outcomes

- Various testing levels are explained with an example of each.
- The major areas of systems testing are identified and executed according to test cases.
- User acceptance is obtained by matching system requirement with the software.
- A structured testing pyramid is developed in accordance with testing requirements.

Testing levels

There are generally four recognized levels of tests: unit testing, integration testing, system testing, and acceptance testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the SWEBOK guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective.

Unit testing

Unit testing, also known as component testing, refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.
These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Unit testing is a software development process that involves synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development lifecycle. Rather than replace traditional QA focuses, it augments it. Unit testing aims to eliminate construction errors before code is promoted to QA; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development and QA process.

Depending on the organization's expectations for software development, unit testing might include static code analysis, data flow analysis, metrics analysis, peer code reviews, code coverage analysis and other software verification practices.

Integration testing

A test level that focuses on interactions between components or systems. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

Acceptance testing

Acceptance testing can mean one of two things:

A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.

- Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT).
- Acceptance testing may be performed as part of the hand-off process between any two phases of development.

System testing

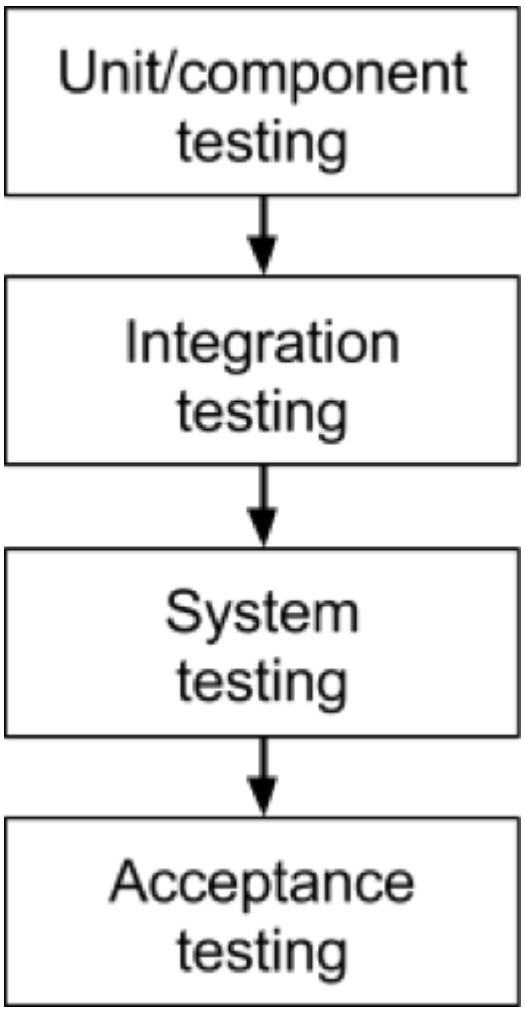
A test level that focuses on verifying that a system as a whole meets specified requirements. For example, a system test might involve testing a logon interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

In addition, the software testing should ensure that the program, as well as working as expected, does not also destroy or partially corrupt its operating environment or cause other processes within that environment to become inoperative (this includes not corrupting shared memory, not consuming or locking up excessive resources and leaving any parallel processes unharmed by its presence).

DIFFERENCES BETWEEN THE DIFFERENT LEVELS OF TESTS

The advantage of detecting any errors in the software early in the day is that by doing Differences are there between the different levels of tests? The focus shifts from early component testing to late acceptance testing. It is important that everybody understands this.

There are generally four recognized levels of tests: unit/component testing, integration testing, system testing, and acceptance testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.



Unit/component testing

The most basic type of testing is unit, or component, testing.

Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate that each individual component is correct in terms of fulfilling requirements and the desired functionality.

This type of testing is performed at the earliest stages of the development process, and in many cases, it is executed by the developers themselves before handing the software over to the testing team.

so, the team minimises software development risks, as well as time and money wasted in having to go back and undo fundamental problems in the program once it is nearly completed.

Integration testing

Integration testing aims to test different parts of the system in combination in order to assess if they work correctly together. By testing the units in groups, any faults in the way they interact together can be identified.

There are many ways to test how different components of the system function at their interface; testers can adopt either a bottom-up or a top-down integration method.

In bottom-up integration testing, testing builds on the results of unit testing by testing higher-level combination of units (called modules) in successively more complex scenarios.

It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

System testing

The next level of testing is system testing. As the name implies, all the components of the software are tested as a whole in order to ensure that the overall product meets the requirements specified.

System testing is a very important step as the software is almost ready to ship and it can be tested in an environment which is very close to that which the user will experience once it is deployed.

System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.

Acceptance testing

Finally, acceptance testing is the level in the software testing process where a product is given the green light or not. The aim of this type of testing is to evaluate whether the system complies with the end-user requirements and if it is ready for deployment.

The testing team will utilise a variety of methods, such as pre-written scenarios and test cases to test the software and use the results obtained from these tools to find ways in which the system can be improved.

The scope of acceptance testing ranges from simply finding spelling mistakes and cosmetic errors, to uncovering bugs that could cause a major error in the application.

By performing acceptance tests, the testing team can find out how the product will perform when it is installed on the user's system. There are also various legal and contractual reasons why acceptance testing has to be carried out.

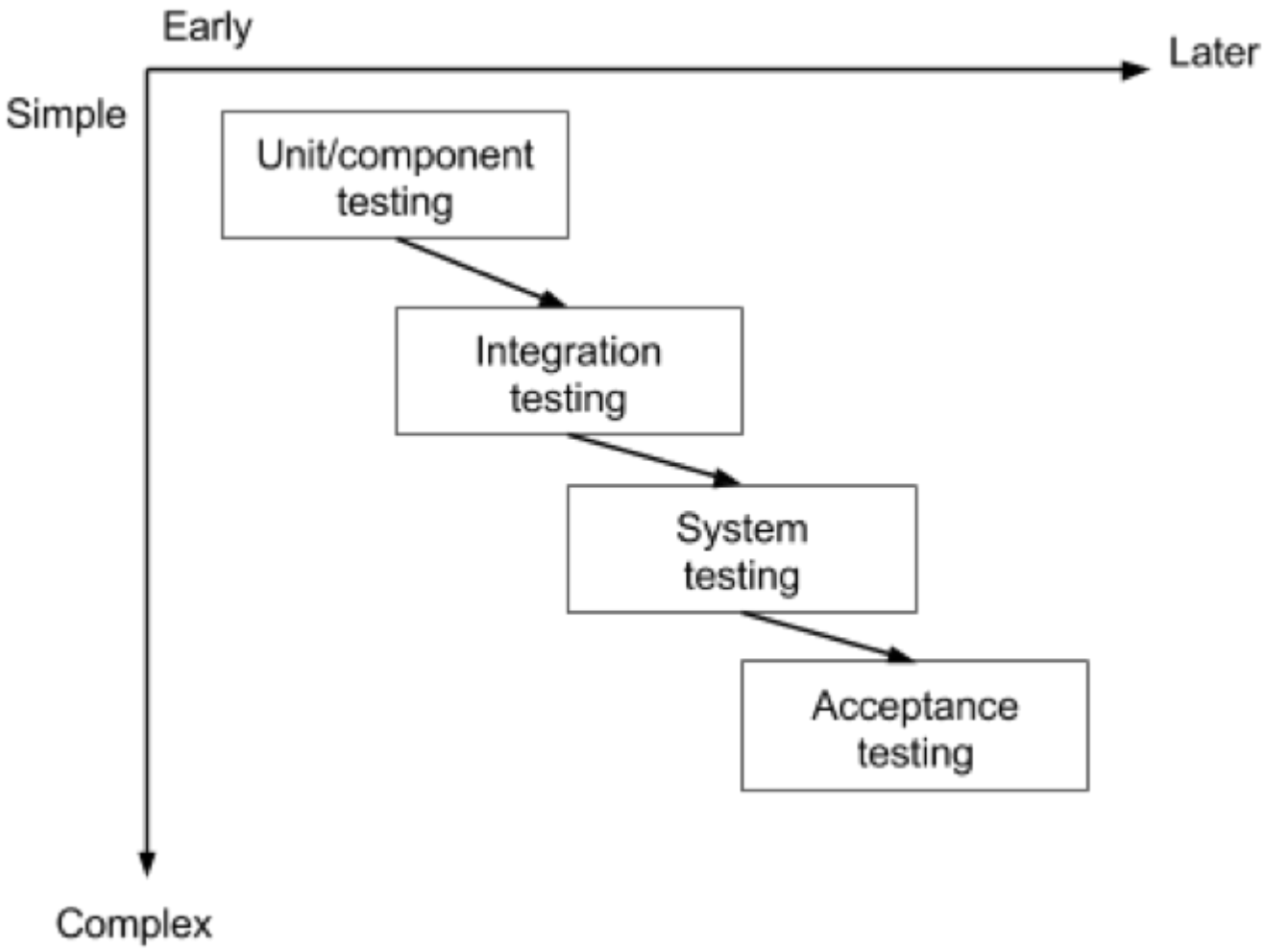
The testing sequences

These four testing types cannot be applied haphazardly during development. There is a logical sequence that should be adhered to in order to minimise the risk of bugs cropping up just before the launch date.

Any testing team should know that testing is important at every phase of the development cycle.

By progressively testing the simpler components of the system and moving on the bigger, more complex groupings, the testers can rest assured they are thoroughly examining the software in the most efficient way possible.

The four levels of tests shouldn't only be seen as a hierarchy that extends from simple to complex, but also as a sequence that spans the whole development process from the early to the later stages. Note however that later does not imply that acceptance testing is done only after say 6 months of development work. In a more agile approach, acceptance testing can be carried out as often as every 2-3 weeks, as a part of the sprint demo. In an organization working more traditionally it is quite typical to have 3-4 releases per year, each following the cycle described here.



- Testing early and testing frequently is well worth the effort.
- By adopting an attitude of constant alertness and scrutiny in all your projects, as well as a systematic approach to testing, the tester can pinpoint any faults in the system sooner, which translates in less time and money wasted later on.
- Detecting software errors early is important since it more effort is needed to fix bugs when the system is nearing launch, and — due to the interactive nature of components in the system — one small bug in a particular component hidden deep within layers of code can result in an effect that is magnified several times over on a system-level.

Prepare a systems test design
Specific Outcome 5

Learning Outcomes

- Test design basics and activities are explained in line with tests to be carried out.
- Deliverables, conditions and procedures are explained in terms of outcomes required.
- Test design specification is prepared and recorded according to organizational requirements
- A systems specification template is created to meet test design specifications.

Steps to Design

There are three fundamental steps you should perform when you have a program to write:

- Define the output and data flows.
- Develop the logic to get to that output.
- Write the program.

Notice that writing the program is the last step in writing the program. This is not as silly as it sounds. Remember that physically building the house is the last stage of building the house; proper planning is critical before any actual building can start. You will find that actually writing and typing in the lines of the program is one of the easiest parts of the programming process. If your design is well thought out, the program practically writes itself; typing it in becomes almost an afterthought to the whole process.

Step 1: Define the Output and Data Flows

Before beginning a program, you must have a firm idea of what the program should produce and what data is needed to produce that output. Just as a builder must know what the house should look like before beginning to build it, a programmer must know what the output is going to be before writing the program.

Anything that the program produces, and the user sees is considered output that you must define. You must know what every screen in the program should look like and what will be on every page of every printed report.

Some programs are rather small, but without knowing where you're heading, you may take longer to finish the program than you would if you first determined the output in detail. Liberty BASIC comes with a sample program called Contact3.bas that you can run. Select File, Open, and select Contact3.bas to load the file from your disk. Press Shift+F5 to run the program.

Step 2: Develop the Logic

After you and the user agree to the goals and output of the program, the rest is up to you. Your job is to take that output definition and decide how to make a computer produce the output. You have taken the overall problem and broken it down into detailed instructions that the computer can carry out. This doesn't mean that you are ready to write the program—quite the contrary. You are now ready to develop the logic that produces that output.

The output definition goes a long way toward describing what the program is supposed to do. Now you must decide how to accomplish the job. You must order the details that you have so they operate in a time-ordered fashion. You must also decide which decisions your program must make, and the actions produced by each of those decisions.

Step 3: Writing the Code

The program writing takes the longest to learn. After you learn to program, however, the actual programming process takes less time than the design if your design is accurate and complete. The nature of programming requires that you learn some new skills. The next few hourly lessons will teach you a lot about programming languages and will help train you to become a better coder so that your programs will not only achieve the goals they are supposed to achieve, but also will be simple to maintain.

Every computer program is built from components, data, and control.

For a single-user application (used by one person at a time), which normally reads data, saves it in a data structure, computes on the data, and writes the results, there is a standard way of organizing the component structure, data structure, and control structure:

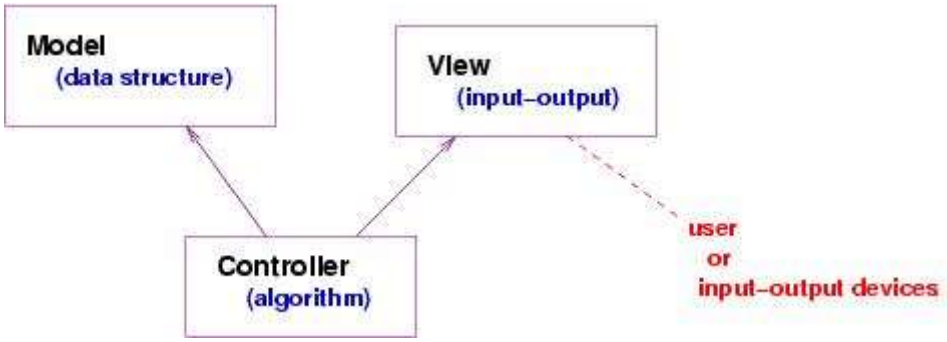
First, design the program's component structure with three components, organized in a model-view-controller pattern.
Next, decide what form of data structure (array, table, set, list, tree, etc.) will hold the program's data. The data structure will be inserted in the program's model component.
Then, write the algorithm that defines the execution steps --- the control structure. The algorithm will be placed inside the program's controller.

Determine the form of input and output (disk file, typed text in a command window, dialogs, a graphical-use interface, etc.) that the program uses. This will be embedded in the program's view.
Once the four-step design is finished, then it is time to convert the design into (Java) coding.
We now consider each stage of the design process.

Component structure

Again, the program's job is to read information into the computer and save it in a format which lets the computer compute answers from the data that can be written.

The program can be written as one large piece of code, but this forces a programmer to think about all parts of the program at once. Years of experience has shown that it is better to design a program in three parts, in a model-view-controller pattern:

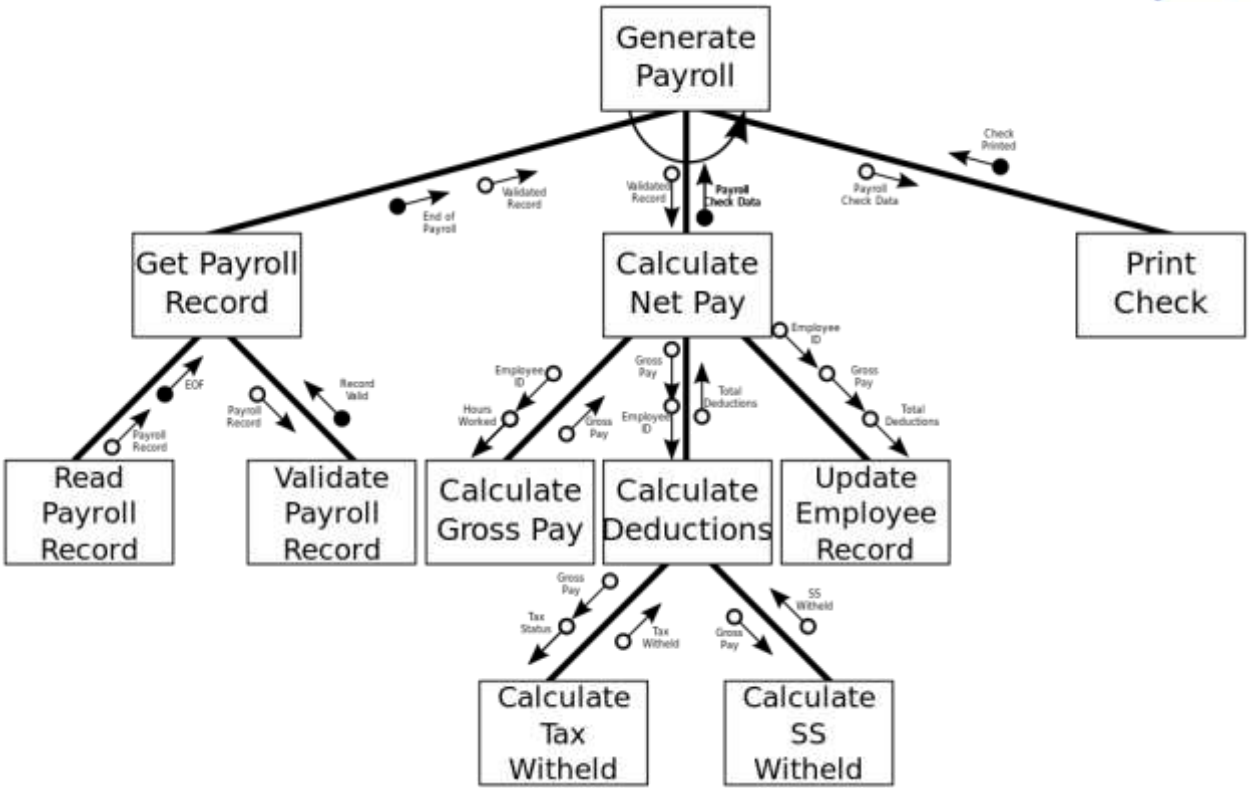


The data flows into the program through the view component and is directed by the controller into the model. The controller component manipulates the data and tells the view to output the answer. A bit more precisely, we have that

- The controller component holds the algorithm, that is the instructions that tell the computer when to read data, compute on it, and write the answers.
- The model component holds the structures that save the data so that it can be easily computed upon. For example, if the program is a spreadsheet program, then the model holds a table that represents the spreadsheet. Or, if the program is the file manager for Linux, then the model is a tree structure that represents the folder-and-file structure of the disk-file system.
- The view component holds the operations that connect the program to the input and output devices.

All three components are important, but the key to building a good quality program is selecting the appropriate data structure for the model component.

A Structure Chart (SC) in software engineering and organizational theory is a chart which shows the breakdown of a system to its lowest manageable levels. They are used in structured programming to arrange program modules into a tree. Each module is represented by a box, which contains the module's name. The tree structure visualizes the relationships between modules.



A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. The lines represent the connection and or ownership between activities and sub activities as they are used in organization charts.

In structured analysis structure charts are used to specify the high-level design, or architecture, of a computer program. As a design tool, they said the programmer in dividing and conquering a large software problem, that is, recursively breaking a problem down into parts that are small enough to be understood by a human brain. The process is called top-down design, or functional decomposition. Programmers use a structure chart to build a program in a manner similar to how an architect uses a blueprint to build a house. In the design stage, the chart is drawn and used as a way for the client and the various software designers to communicate. During the actual building of the program, the chart is continually referred to as the masterplan.

A structure chart depicts

- the size and complexity of the system, and
- number of readily identifiable functions and modules within each function and
- whether each identifiable function is a manageable entity or should be broken down into smaller components.

The **flow chart** is a means to visually present the flow of data through an information processing system, the operations performed within the system and the sequence in which they are performed. In this lesson, we shall concern ourselves with the program flow chart, which describes what operations (and in what sequence) are required to solve a given problem. The program flow chart can be likened to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flow chart prior to writing a computer program.

FLOW CHART OBJECTIVES

At the end of this lesson, you will be able to understand:

- the meaning of flow chart
- the basic parts of the flow chart such as flow chart symbols and the flow lines connecting these symbols.
- the advantages and limitations of flowchart

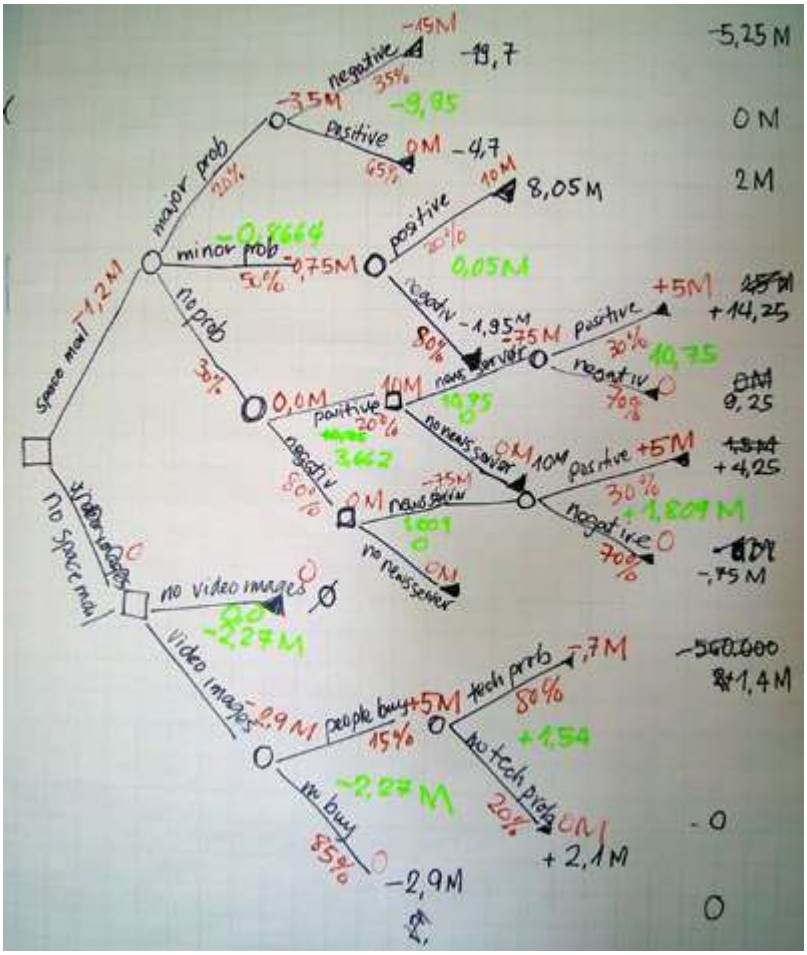
FLOW CHART SOFTWARE

draw Flow Chart Software will help the designer create professional basic flow chart, business process modeling notation chart, cross functional flow chart, data flow diagram, list and workflow chart from examples - with no drawing required.

Flowchart Templates

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

Decision trees are commonly used in operations research, specifically in decision analysis, to help identify a strategy most likely to reach a goal but are also a popular tool in machine learning.



Traditionally, decision trees have been created manually.

A decision tree is a flowchart-like structure in which each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label. The paths from root to leaf represent classification rules.

In decision analysis, a decision tree and the closely related influence diagram are used as a visual and analytical decision support tool, where the expected values of competing alternatives are calculated.

A decision tree consists of three types of nodes:

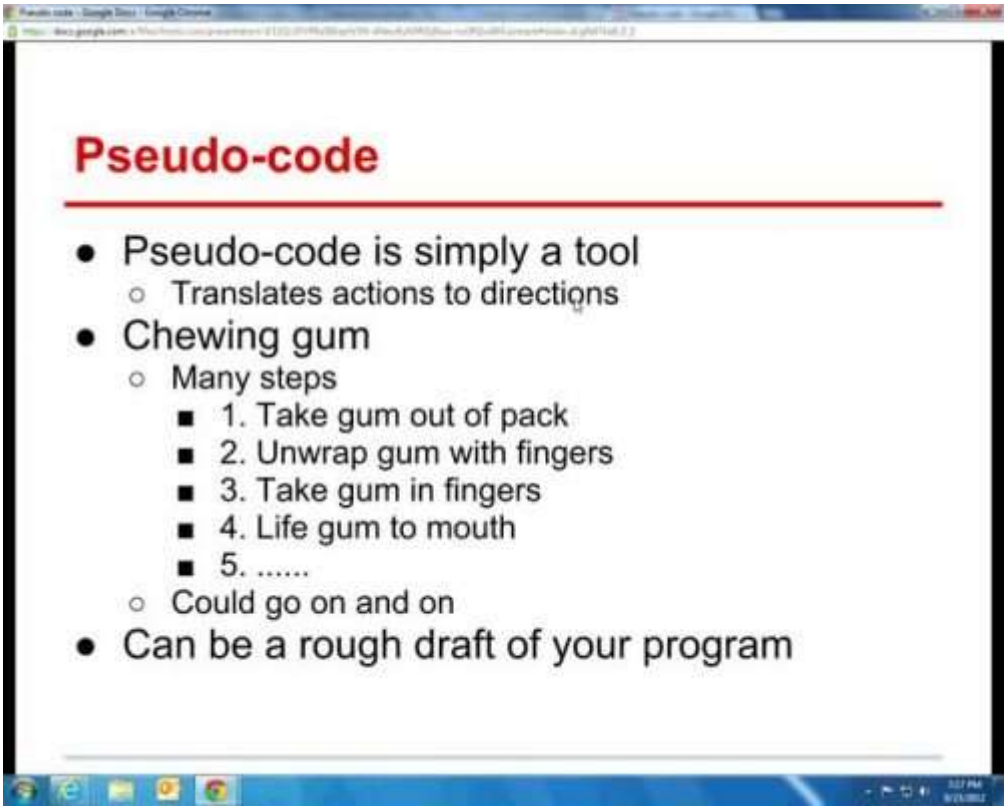
- Decision nodes – typically represented by squares
- Chance nodes – typically represented by circles
- End nodes – typically represented by triangles

Decision trees are commonly used in operations research and operations management. If, in practice, decisions have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model or online selection model algorithm. Another use of decision tree is as a descriptive means for calculating conditional probabilities.

Decision trees, influence diagrams, utility functions, and other decision analysis tools and methods are taught to undergraduate students in schools of business, health economics, and public health, and are examples of operations research or management science methods.

Pseudo-code is a simple way of writing programming code in English. Pseudo-code is not actual programming language. It uses short phrases to write code for programs before you actually create it in a specific language. Once you know what the program is about and how it will function, then you can use pseudo-code to create statements to achieve the required results for your program.

Pseudo-code makes creating programs easier. Programs can be complex and long; preparation is the key. For years, flowcharts were used to map out programs before writing one line of code in a language. However, they were difficult to modify and with the advancement of programming languages, it was difficult to display all parts of a program with a flowchart. It is challenging to find a mistake without understanding the complete flow of a program. That is where pseudo-code becomes more appealing.



To use pseudo-code, all you do is write what you want your program to say in English. Pseudo-code allows you to translate your statements into any language because there are no special commands and it is not standardized. Writing out programs before you code can enable you to better organize and see where you may have left out needed parts in your programs. All you have to do is write it out in your own words in short statements.

A decision table is considered balanced or complete if it includes every possible combination of input variable. In other words, balanced decision tables prescribe an action in every situation where the input variables are provided.

Software Testing Life Cycle

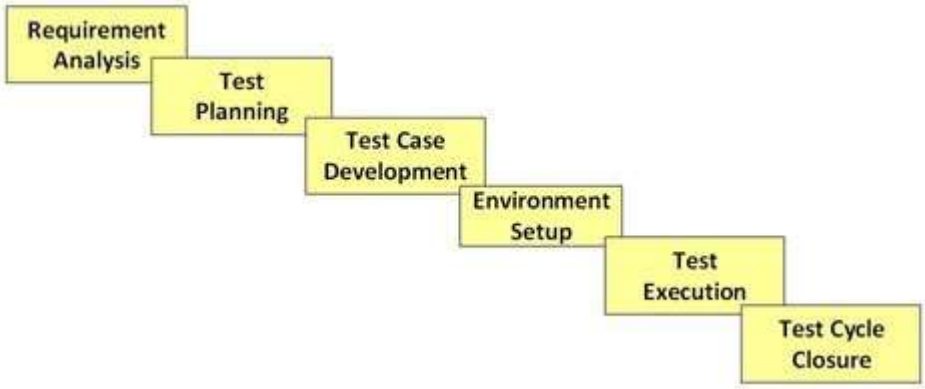
Contrary to popular belief, Software Testing is not a just a single activity.

What is Software Testing Life Cycle (STLC)?

Software Testing Life Cycle (STLC) is defined as a sequence of activities conducted to perform Software Testing.

It consists of series of activities carried out methodologically to help certify your software product.

Diagram - Different stages in Software Test Life Cycle



Each of these stages have a definite Entry and Exit criteria; , Activities & Deliverables associated with it.

What is Entry and Exit Criteria

Entry Criteria: Entry Criteria gives the prerequisite items that must be completed before testing can begin.

Exit Criteria: Exit Criteria defines the items that must be completed before testing can be concluded

You have Entry and Exit Criteria for all levels in the Software Testing Life Cycle (STLC)

In an Ideal world you will not enter the next stage until the exit criteria for the previous stage is met. But practically this is not always possible. So, for this tutorial, we will focus on activities and deliverables for the different stages in STLC life cycle. Let's look into them in detail.

Requirement Analysis

During this phase, test team studies the requirements from a testing point of view to identify the testable requirements.

The QA team may interact with various stakeholders (Client, Business Analyst, Technical Leads, System Architects etc) to understand the requirements in detail.

Requirements could be either Functional (defining what the software must do) or Non-Functional (defining system performance /security availability)

Automation feasibility for the given testing project is also done in this stage. Activities

- Identify types of tests to be performed.
- Gather details about testing priorities and focus.
- Prepare Requirement Traceability Matrix (RTM).
- Identify test environment details where testing is supposed to be carried out.
- Automation feasibility analysis (if required).

Deliverables

RTM

Automation feasibility report. (if applicable)

Test Planning

Typically, in this stage, a Senior QA manager will determine effort and cost estimates for the project and would prepare and finalize the Test Plan. In this phase, Test Strategy is also determined.

Activities

- Preparation of test plan/strategy document for various types of testing
- Test tool selection
- Test effort estimation
- Resource planning and determining roles and responsibilities.
- Training requirement

Deliverables

- Test plan /strategy document.
- Effort estimation document.
- Test Case Development

This phase involves creation, verification and rework of test cases & test scripts. Test data, is identified/created and is reviewed and then reworked as well. Activities

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (If Test Environment is available)

Deliverables

Test cases/scripts

Test data

Test Environment Setup

Test environment decides the software and hardware conditions under which a work product is tested. Test environment set-up is one of the critical aspects of testing process and can be done in parallel with Test Case Development Stage. Test team may not be involved in this activity if the customer/development team provides the test environment in which case the test team is required to do a readiness check (smoke testing) of the given environment.

Activities

Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.

- Setup test Environment and test data
- Perform smoke test on the build

Deliverables

- Environment ready with test data set up
- Smoke Test Results.

Test Execution

During this phase the testers will carry out the testing based on the test plans and the test cases prepared. Bugs will be reported back to the development team for correction and retesting will be performed. Activities

- Execute tests as per plan
- Document test results, and log defects for failed cases
- Map defects to test cases in RTM
- Retest the Defect fixes
- Track the defects to closure

Deliverables

Completed RTM with execution status

Test cases updated with results

Defect reports

Test Cycle Closure

Testing team will meet, discuss and analyse testing artifacts to identify strategies that have to be implemented in future, taking lessons from the current test cycle. The idea is to remove the process bottlenecks for future test cycles and share best practices for any similar projects in future. Activities

- Evaluate cycle completion criteria based on Time, Test coverage, Cost, Software, Critical Business Objectives, Quality
- Prepare test metrics based on the above parameters.
- Document the learning out of the project
- Prepare Test closure report
- Qualitative and quantitative reporting of quality of the work product to the customer.
- Test result analysis to find out the defect distribution by type and severity.

Perform software testing
Specific Outcome 6

Learning Outcomes

- Test requirements are defined and recorded in terms of the functionality of the software.
- Tests and test cases are designed to meet software requirements.
- The software test process is defined in accordance the specification of the test activities.
- Tools are selected to support the test process.
- Testing is conducted in accordance with the test plan.

Testing Requirements

Software testing is not an activity to take up when the product is ready. An effective Software Testing begins with a proper plan from the user requirements stage itself. Software testability is the ease with which a computer program is tested. Metrics can be used to measure the testability of a product. The requirements for effective Software Testing are given in the following sub-sections.

Operability in Software Testing:

- The better the software works, the more efficiently it can be tested.
- The system has few bugs (bugs add analysis and reporting overhead to the test process)
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development & testing)

Observability in Software Testing:

- What is seen is what is tested
- Distinct output is generated for each input
- System states and variables are visible or queriable during execution
- Past system states and variables are visible or queriable (eg., transaction logs)
- All factors affecting the output are visible
- Incorrect output is easily identified
- Incorrect input is easily identified
- Internal errors are automatically detected through self-testing mechanism
- Internally errors are automatically reported
- Source code is accessible

Controllability in Software Testing:

- The better the software is controlled, the more the testing can be automated and optimised.
- All possible outputs can be generated through some combination of input in Software Testing
- All code is executable through some combination of input in Software Testing
- Software and hardware states can be controlled directly by testing
- Input and output formats are consistent and structured in Software Testing
- Tests can be conveniently specified, automated, and reproduced.

Decomposability in Software Testing:

- By controlling the scope of testing, problems can be isolated quickly, and smarter testing can be performed.
- The software system is built from independent modules
- Software modules can be tested independently in Software Testing

Simplicity in Software Testing:

- The less there is to test, the more quickly it can be tested in Software Testing
- Functional simplicity
- Structural simplicity
- Code simplicity

Stability in Software Testing:

- The fewer the changes, the fewer the disruptions to testing
- Changes to the software are infrequent
- Changes to the software are controlled in Software Testing
- Changes to the software do not invalidate existing tests in Software Testing
- The software recovers well from failures in Software Testing

Understandability in Software Testing:

- The more information we have, the smarter we will test
- The design is well understood in Software Testing
- Dependencies between internal external and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible
- Technical documentation is well organized in Software Testing
- Technical documentation is specific and detailer
- Technical documentation is accurate

In designing of the program, a data structure is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. In comparison, a data structure is a concrete implementation of the space provided by an ADT.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways.

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analysed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost

Database design is the process of producing a detailed data model of a database. This data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a data definition language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity.

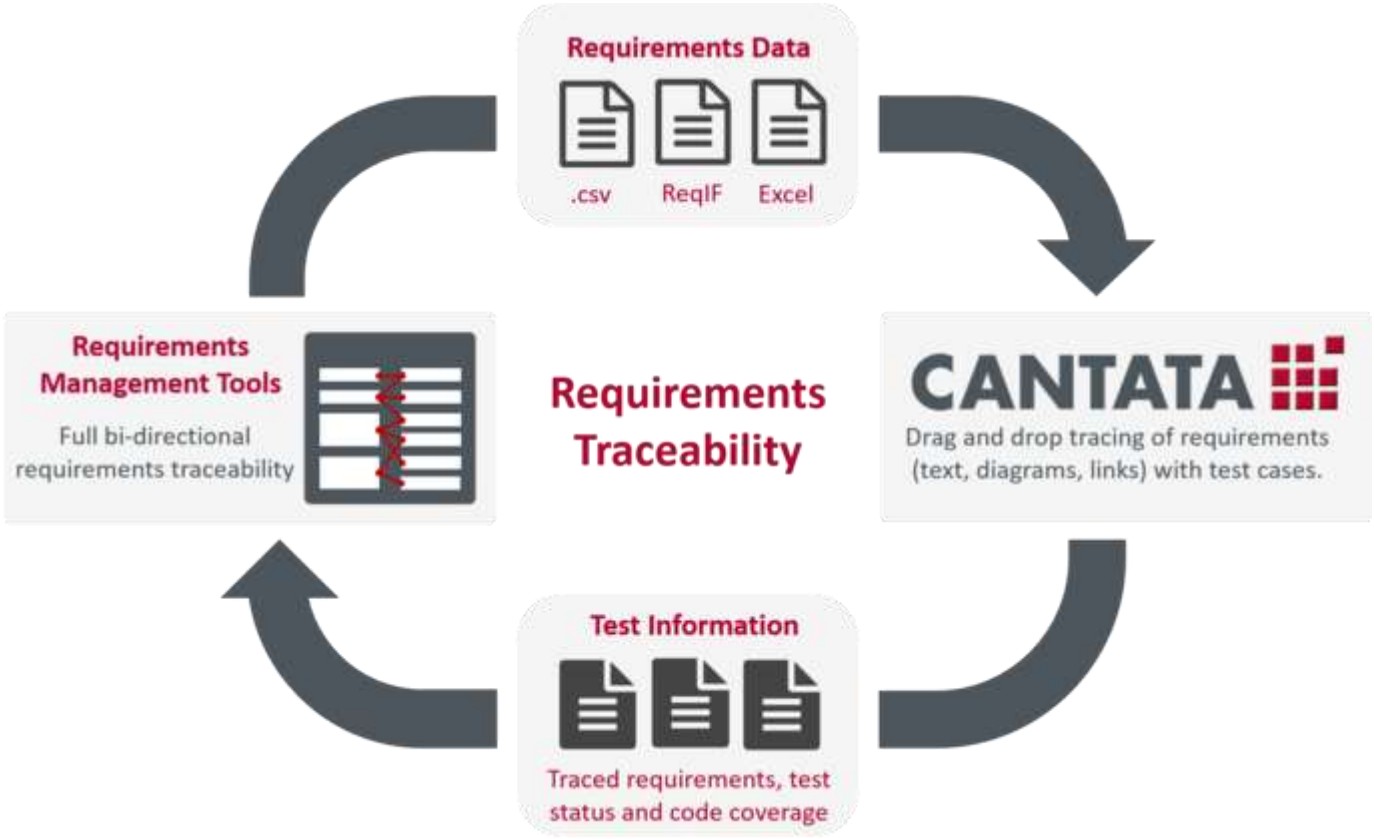
The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data.

In the relational model these are the tables and views. In an object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the database management system (DBMS).

The process of doing database design generally consists of a number of steps which will be carried out by the database designer. Usually, the designer must:

- Determine the data to be stored in the database.
- Determine the relationships between the different data elements.
- Superimpose a logical structure upon the data on the basis of these relationships.
- Within the relational model the final step above can generally be broken down into two further steps, that of determining the grouping of information within the system, generally determining what are the basic objects about which information is being stored, and then determining the relationships between these groups of information, or objects.

Cantata capabilities by test technique



REQUIREMENTS DRIVEN TESTING

Cantata supports requirements driven testing to verify that the software does what it should do (derived from low level requirements, specifications or models).

The Cantata test harness drives the code under test via a test script calling a function with selected inputs (parameters and data) and automating the checking of actual against expected behavior.

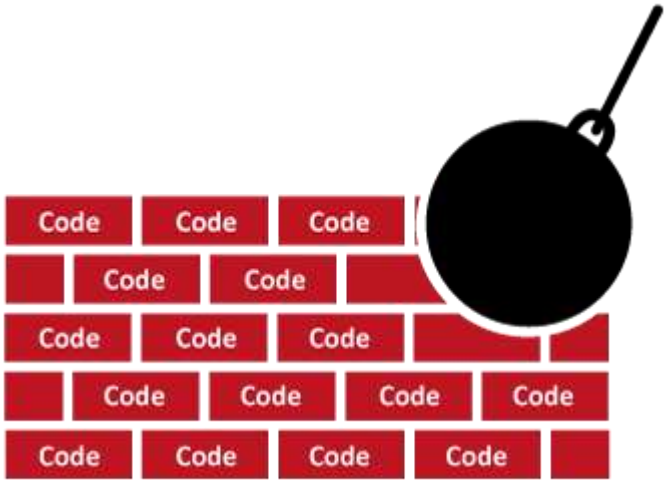
Requirements imported into the Cantata IDE make it easier to define test cases and trace requirements with test scripts or test cases.

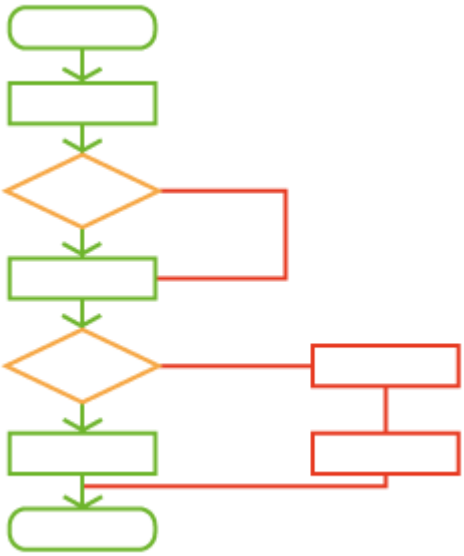
ROBUSTNESS TESTING

Cantata supports dynamic robustness testing to verify that code copes with abnormal conditions using techniques recommended by all software safety standards such as:

- abnormal system initialization
- invalid inputs & data
- failure mode handling
- boundary values
- arithmetic edge conditions
- timing constraints
- memory allocation
- error seeding

Robustness testing is made easy with Cantata Robustness Rule Sets of pre-defined values for basic data types, in looping test cases. All accessible global data is also automatically checked for inadvertent changes. Cantata test scripts and wrapping make it easy to create abnormal test pre-conditions and inject errors during test execution.





STRUCTURAL TESTING

Structural testing identifies source code untested by requirements driven and robustness tests and addresses those gaps.

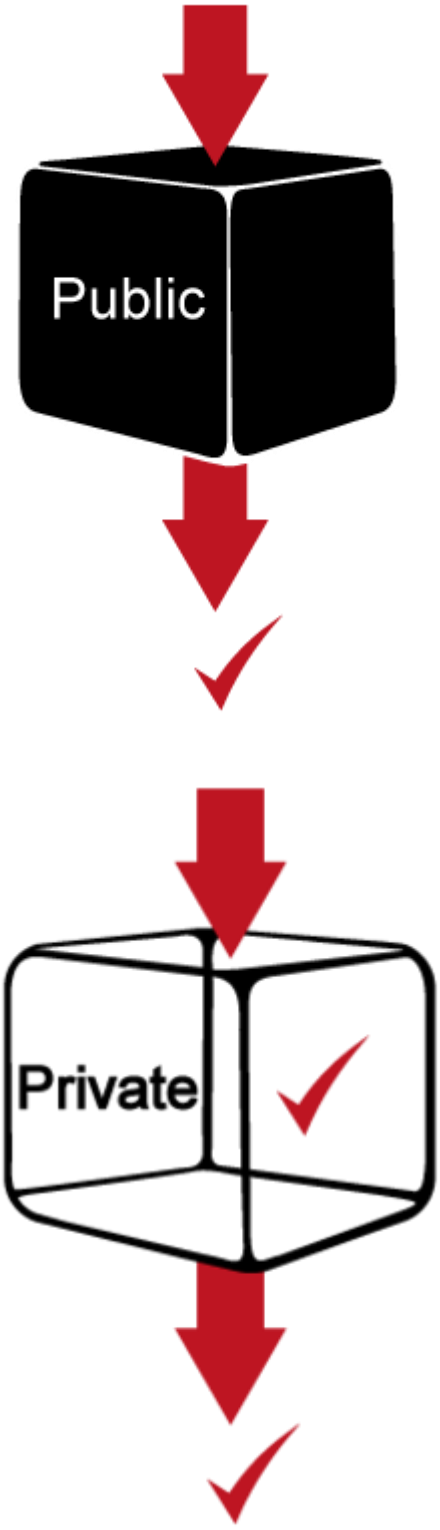
Cantata code coverage (used within or outside Cantata tests) pinpoints any such unexecuted code elements in specific or all contexts. Gaps can be resolved by adding new requirements-based test cases, documenting reasons why the code should not be executed in a particular context, or removing redundant code.

BLACK BOX TESTING

Black-box testing verifies code only via its public interfaces and does not require knowledge of internal code implementation for checking correct behavior.

Cantata brings power to this technique with user-selected or pre-defined parameterised looping tests, so the same functions can be called with large data sets of input parameter or data values. The sequence of expected calls and expected return values can vary according to the input data sets, alongside global data outputs checked throughout the test run.

Cantata table driven test data sets can also be imported/exported via CSV. A GUI combinatorial effect calculator aids test vector selection. Cantata automatic coverage optimisation identifies the precise set of test case vectors needed to reach coverage targets.



WHITE BOX TESTING

White-box testing verifies code by granting access to the encapsulated code internals.

Cantata grants testers direct access to these without polluting production code with conditional compilation, by using fully automatic accessibility instrumentation only during testing. Test scripts can

directly call private or static functions and set / check parameters and data declared private or static to a file or local function.

Cantata white-box tests can set up trigger conditions, and directly check values or internal behaviour more efficiently than with black-box tests.

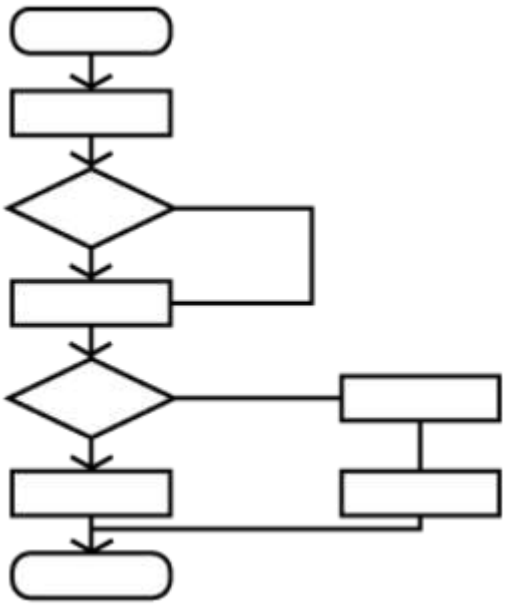
Cantata call control can also be used on calls inside the compilation boundary for white-box testing.

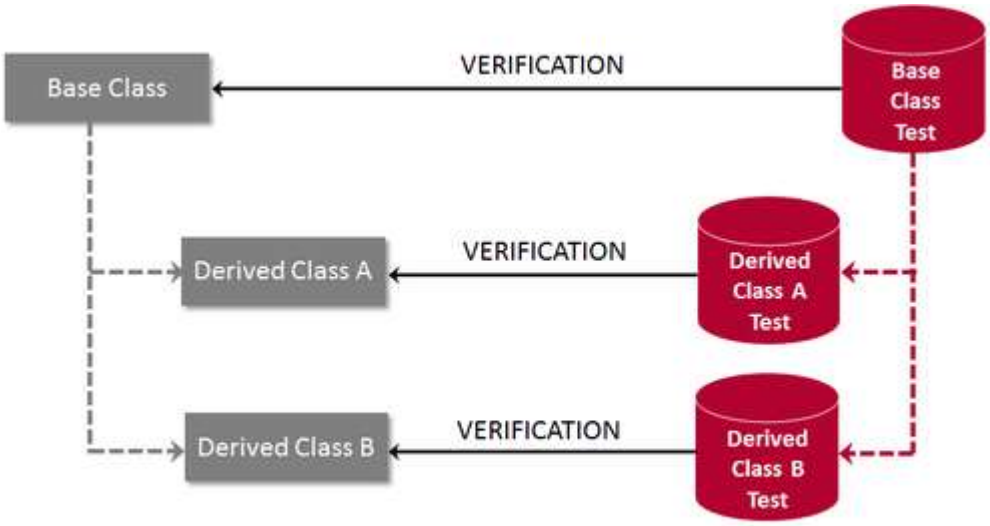
PROCEDURAL AND STATE MACHINE TESTING

Cantata procedural tests verify the processing logic or algorithms in functions using appropriate sets of parameter / data inputs, and checking that actual call order, data read/writes and return values are as expected. Cantata parses the source code to generate test scripts with the necessary test hooks on function parameters and accessible data for users to set and check.

State machines are tested by setting trigger conditions and creating events, to verify correct state transitions. Cantata white-box testing makes this particularly efficient through direct access to local function static data storing the machine state.

Cantata user defined context code coverage can also be used to verify code has been tested in different states.





Parallel Inheritance Hierarchy

OBJECT ORIENTED TESTING

Cantata OO style tests are implemented as classes for testing methods, templates, or clusters of classes. They feature automated:

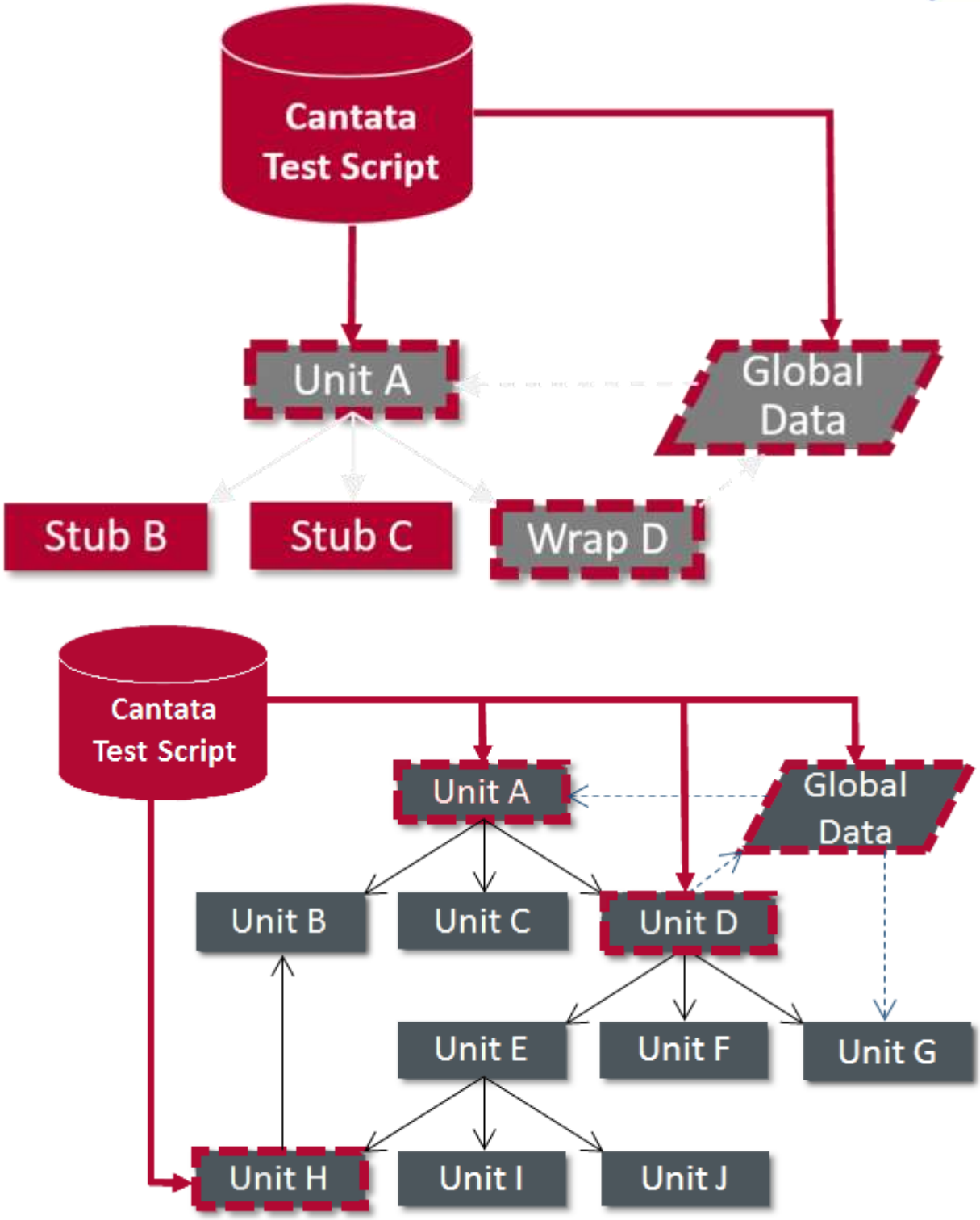
- test case re-use via a parallel inheritance hierarchy
- test class inheritance structure for inherited classes
- concrete implementation of abstract base classes (ABCs) or pure virtual methods (PVMs)

To break the class dependency chain and make C++ isolation testing easy, Cantata automatically finds and resolves dependencies on undefined references that are not directly called by the software under test.

ISOLATION UNIT TESTING

Cantata does not dictate what code item is tested as a unit, or how much each unit is isolated (decoupled) from the rest of the system. Complete flexibility is given by automatic configurable generation of stubs and isolates to simulate, and wrappers to intercept calls made both externally and internally by the unit under test. The correct internal processing and correct use of interfaces (parameters and data), can be verified with Cantata as an isolated executable as soon as a unit compiles, without the need for the rest of the system.

Cantata generates an initial executable test case for each function / method in the selected source file(s) under test. Cantata unit tests call a function, initializing all input / output parameters, accessible data and expected call order. Test cases are independent of each other and the platform on which they run, avoiding a daisy-chain of dependencies.



INTEGRATION TESTING

Cantata driven integration tests are just large unit tests, but with a cluster of multiple units built as a single executable. Such a tests can verify the same internal behavior of unit, but more importantly the interactions between units such as:

- inter-unit memory allocation
- order of data read / writes
- order of calls between units

Cantata integration tests provide the power to choose test script driver entry-points, with both internal and external simulations (stubs and isolates) or interceptions (wrappers) for flexible use of top-down, bottom-up or all-as-one integration testing styles. Cantata wrappers can also be used where Cantata is not the test diver, adding further control over integration tests

---

The techniques set out above may be applied in various combinations to comply with the requirements of software safety standards and industry best practice. For more detailed information on designing test cases and the requirements of a specific standard, please see our extensive set of resources.

Rational Performance Tester

Rational Performance Tester is a tool for automated performance testing of web- and server-based applications from the Rational Software division of IBM. It allows users to create tests that mimic user transactions between an application client and server. During test execution, these transactions are replicated in parallel to simulate a large transaction load on the server. Server response time measurements are collected to identify the presence and cause of any potential application bottlenecks. It is primarily used by Software Quality Assurance teams to perform automated software performance testing.

Performance test creation process

The following is an overview of the process of system performance validation with RPT.

Test creation

Tests are created using the RPT recording mechanism. The RPT recorder captures all transactions between application client (such as a web browser) and an application server. The resulting test is displayed as a tree view, where each branch of the tree represents a browser or client request and response.

Test editing

RPT tests are not represented as code. RPT tests are represented as a tree view, where each branch of the tree represents a browser or client request and response. To edit the test, the user selects menu options which allow for insertion of loops, "if-then" type decision structures, and response verification. Should a custom coding solution be required, users can insert Java code modules to perform complex computations.

RPT performs automatic test editing for data pooling and data correlation. With data pooling, RPT automatically edits tests to separate test data from the test actions. Test data, such as a user login ID and password values that were typed in by the user during the test, are stored in a spreadsheet like data pool. The test is crafted in such a way so that during test playback, each simulated user will access one row of the test data from the data pool. This ensures that each unique user uses unique data during playback, and prevents a situation where, for example, 100 simulated users attempt to log in with a single user ID and password.

Data correlation is the process by which RPT ensures continuity between test actions. Often in a performance test a value is created at one step of a test and subsequently used during a later step. Whenever such a value is created, RPT stores that value in a variable, and uses that variable later on the test when the data is accessed. This prevents hard coding of values and ensures that tests will play back correctly in dynamic data environments.

Test scheduling

RPT can execute a single test, or it can create a suite of tests for playback. When creating a suite of tests, users drag and drop tests onto a schedule to simulate a given sequence of events. Tests can be grouped by user profile, to represent the activities of different types of users on a system. The volume of transactions can be set to increase at defined intervals to increase load during a test, in an effort to identify system bottlenecks.

Test reporting

RPT offers a variety of reports to identify the presence and cause of system performance bottlenecks. There are reports that measure accuracy of system response which ensure that the system did not error out or crash during a test. There are reports to measure system performance metrics such as disk, network and CPU utilization, to identify the presence of hardware bottlenecks. To identify software related bottlenecks, there are reports to measure load, throughput and response times. For Java EE environments, when a bottleneck is identified, users can drill down on the performance reports to identify the cause of the bottleneck, identifying slow performance classes, methods and individual lines of application source code.

Rational Functional Tester

Rational Functional Tester is a tool for automated testing of software applications from the Rational Software division of IBM. It allows users to create tests that mimic the actions and assessments of a human tester. It is primarily used by Software Quality Assurance teams to perform automated regression testing.

Storyboard Testing

Introduced in version 8.1 of Rational Functional Tester, this technology enables testers to edit test scripts by acting against screen shots of the application.

Object

The Rational Functional Tester Object Map is the underlying technology used by Rational Functional Tester to find and act against the objects within an application. The Object Map is automatically created by the test recorder when tests are created and contains a list of properties used to identify objects during playback.

Script Assure

During playback, Rational Functional Tester uses the Object Map to find and act against the application interface. However, during development it is often the case that objects change between the time the script was recorded and when a script was executed. Script Assure technology enables Rational Functional Tester to ignore discrepancies between object definitions captured during recording and playback to ensure that test script execution runs uninterrupted. Script Assure sensitivity, which determines how big an object map discrepancy is acceptable, is set by the user.

Data Driven Testing

It is common for a single functional regression test to be executed multiple times with different data. To facilitate this, the test recorder can automatically parametrize data entry values, and store the data in a spreadsheet like data pool. This enables tester to add additional test data cases to the test data pool without having to modify any test code. This strategy increases test coverage and the value of a given functional test.

Dynamic Scripting Using Find API

Rational Functional Test script, Eclipse Integration uses Java as its scripting language. The Script is a .java file and has full access to the standard Java APIs or any other API exposed through other class libraries.

Apart from this RFT itself provides a rich API to help user further modify the script generated through the recorder. Rational Test Script class that is the base class for any Test Script provides a find API that can be used to find the control based on the given properties.

Test & Performance Tools Platform

The Test & Performance Tools Platform (TPTP) is an Eclipse tool used to profile plug-ins of the IDE that may run on different platforms. TPTP is tightly integrated into Eclipse so that it can make the profiling from within the IDE. It is used to find and isolate performance problems. Such problems can be performance bottlenecks, object leaks, or system resource limits. It can be used with both simple and complex applications, like stand-alone applications, plug-ins, or multi-machine enterprise applications.

As TPTP is integrated into the Eclipse project, it can be easily extended.

TPTP will not be available in releases of Eclipse 3.7 (Indigo) as it undergoes archival from the Eclipse project.