Classroom Management (Software Verification and Validation)

Table of Contents:

Lesson 01

Introduction to Software Testing

Topic- 001:

Software testing is an important part of software development as it is helpful for detecting errors and bugs in a software system. Software testing helps validate the software system whether it works as expected or not. Test cases are implemented to check that the system which is to be tested is working as specified in the requirement documents or not. It is estimated that software testing consumes almost 50% of the total software development efforts. Software testing shows presence of errors and bugs in a software system but it cannot prove absence of errors. Software testing has always been an important technique to validate the system under test.

Software testing with specific reference to verification and validation deserves a discussion on definition with a couple of examples.

In this lecture, the following topics are discussed:

- Introduction.
- Overall outline of the course

Topic- 002:

Students need to be motivated why we need to do software testing; this needs examples where software systems failed due to lack of testing. Another aspect is the increasing size of software and pressing time-to-market limitations.

In this lecture, the following topics are discussed:

- Motivation for software testing
- Sources of problems

Topic- 003:

Students once motivated require a formal definition of testing and reliability. Software testing terms at the first level are introduced.

In this lecture, the following topics are discussed:

- Working definition of reliability and software testing:
  Reliability testing is a type of software testing that helps in checking whether software or application can perform its operations without encountering any failures in a particular

environment and a specific period of time. Software testing can be defined as a method that allows us to check whether the software product or application matches the expected requirements and it also ensures that the software product is error/defect free.

- What is a software fault, error, bug, failure or debugging?
  A software error can be defined as a mistake made when coding a program. For example, semi-colon missing from a line of code in C++ will result in a syntax error. When the mistake or defect has been identified by the development team then the defect is called a bug. A failure occurs when a program or software fails to meet its requirements in the build phase. Debugging can be defined as the process of finding and removing of errors from software or application.

Topic- 004:

Students need to understand how software testing relates to software development in a lifecycle manner such that we are aware when do we do software testing and why there is a requirement to do testing in a continual manner.

In this lecture, the following topics are discussed:

- Software testing and Software lifecycle
- Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps, or missing requirements in contrast to actual requirements. Software lifecycle is a process that produces software with the highest quality and lowest cost in the shortest time. It includes a detailed plan for how to develop, alter, maintain, and replace a software system. It involves several distinct stages, including planning, design, building, testing, and deployment. Popular SDLC models include the waterfall model, spiral model, and agile model. SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace, and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

- Software testing myths
  - Some of the testing myths include the following:
    - Testing is Too Expensive
    - Testing is Time-Consuming
    - Only Fully Developed Products Tested
    - Complete Testing is Possible

        - A Tested Software is Bug-Free.
        - Missed Defects are due to Testers
        - Testers are Responsible for Quality of Product
        - Test Automation should use wherever possible to Reduce Time
        - Anyone can Test a Software Application
        - A Tester's only Task is to Find Bugs
- Goals and Limitations of Testing:
- Goals include the following:
    - Verification and Validation: Testing is a quality control measure used to verify that a product works as desired. Software testing provides a status report of the actual product in comparison to product requirements (written and implicit). Testing process must verify and validate whether the software fulfills conditions laid down for its release/use.
    - Priority Coverage: Exhaustive testing is impossible. We should perform tests efficiently and effectively, within budgetary and scheduling limitations. Therefore, testing needs to assign effort reasonably and prioritize thoroughly.
    - Balanced testing process: Testing process must balance the written requirements, real-world technical limitations, and user expectations. The testing process and its results must be repeatable and independent of the tester, i.e., consistent, and unbiased.
    - Traceable process: Documenting both the successes and failures helps in easing the process of testing. What was tested, and how it was tested, are needed as part of an ongoing testing process.
    - Deterministic process: Problem detection should not be random in testing. We should know what we are doing, what are we targeting, what will be the possible outcome
- Limitations include the following:
    - We cannot test a program completely
    - We can only test against system requirements
    - Exhaustive (total) testing is impossible in present scenario.
    - Time and likewise budget constraints normally require very careful planning of the testing effort.
    - Compromise between thoroughness and budget.
    - Test results used to make business decisions for release dates.
    - Even if you do find the last bug, you will never know it
    - You will run out of time before you run out of test cases
    - You cannot test every path
    - Every valid input cannot test
    -

Lesson 02

Software quality control and quality assurance

Topic- 005:

Software quality definition considering software quality attributes is provided. The students are introduced with default document for quality which is the software requirements specification document (SRS).

In this lecture, the following topics are discussed:

- Software quality attributes
    - Software Quality Attributes are: Correctness, Reliability, Adequacy, Learnability, Robustness, Maintainability, Readability, Extensibility, Testability, Efficiency, Portability.
    - Correctness: The correctness of a software system refers to:
        - Agreement of program code with specifications
          – Independence of the actual application of the software system.
        - The correctness of a program becomes especially critical when it is embedded in a complex software system.
    - Reliability: Reliability of a software system derives from
    - Availability
        - The behavior over time for the fulfillment of a given specification depends on the reliability of the software system.
        - Reliability of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).
        - A software system can be seen as reliable if this test produces a low error rate (i.e., the probability that an error will occur in a specified time interval.)
        - The error rate depends on the frequency of inputs and on the probability that an individual input will lead to an error.
    - Adequacy: Factors for the requirement of Adequacy:
        - The input required of the user should be limited to only what is necessary. The software system should expect information only if it is necessary for the functions that the user wishes to carry out. The software system should enable flexible data input on the part of the user and should carry out plausibility checks on the input. In dialog-driven software systems, we

vest particular importance in the uniformity, clarity and simplicity of the dialogs.

    - The performance offered by the software system should be adapted to the wishes of the user with the consideration given to extensibility; i.e., the functions should be limited to these in the specification.
    - The results produced by the software system: The results that a software system delivers should be output in a clear and well-structured form and be easy to interpret. The software system should afford the user flexibility with respect to the scope, the degree of detail, and the form of presentation of the results. Error messages must be provided in a form that is comprehensible for the user.
- Learnability: Learnability of a software system depends on:
    - The design of user interfaces
      The clarity and the simplicity of the user instructions (tutorial or user manual).
    - The user interface should present information as close to reality as possible and permit efficient utilization of the software's failures.
    - The user manual should be structured clearly and simply and be free of all dead weight. It should explain to the user what the software system should do, how the individual functions are activated, what relationships exist between functions, and which exceptions might arise and how they can be corrected. In addition, the user manual should serve as a reference that supports the user in quickly and comfortably finding the correct answers to questions.
- Robustness: Robustness reduces the impact of operational mistakes, erroneous input data, and hardware errors. A software system is robust if the consequences of an error in its operation, in the input, or in the hardware, in relation to a given application, are inversely proportional to the probability of the occurrence of this error in the given application.
    - Frequent errors (e.g. erroneous commands, typing errors) must be handled with particular care.
    - Less frequent errors (e.g. power failure) can be handled more laxly, but still must not lead to irreversible consequences.
- Maintainability: Maintainability = suitability for debugging (localization and correction of errors) and for modification and extension of functionality.
    - The maintainability of a software system depends on its:
    - – Readability
      – Extensibility
      – Testability
    - Readability: Readability of a software system depends on its:

    - – Form of representation
      – Programming style
      – Consistency
      – Readability of the implementation programming languages
      – Structuredness of the system
      – Quality of the documentation
      – Tools available for inspection
- Extensibility: Extensibility allows required modifications at the appropriate locations to be made without undesirable side effects. Extensibility of a software system depends on its:
    - – Structuredness (modularity) of the software system
      – Possibilities that the implementation language provides for this purpose
      – Readability (to find the appropriate location) of the code
      – Availability of comprehensible program documentation
- Testability: suitability for allowing the programmer to follow program execution (runtime behaviour under given conditions) and for debugging. The testability of a software system depends on its:
    - – Modularity
      – Structuredness
    - Modular, well-structured programs prove more suitable for systematic, stepwise testing than monolithic, unstructured programs.
    - Testing tools and the possibility of formulating consistency conditions (assertions) in the source code reduce the testing effort and provide important prerequisites for the extensive, systematic testing of all system components.
    - Efficiency: ability of a software system to fulfil its purpose with the best possible utilization of all necessary resources (time, storage, transmission channels, and peripherals).
- Portability: the ease with which a software system can be adapted to run on computers other than the one for which it was designed.
    - The portability of a software system depends on:
    - – Degree of hardware independence
      – Implementation language
      – Extent of exploitation of specialized system functions
      – Hardware properties
      – Structuredness: System-dependent elements are collected in easily interchangeable program components.
    - A software system can be said to be portable if the effort required for porting it proves significantly less than the effort necessary for a new implementation.

- Default document for quality:
    - Default document or the starting point for quality control or software testing is the system specification document. This is an artifact that is written in several forms that include use case descriptions, user stories, software or system requirement specification documents etc.
- Functional and non-functional aspects of system vs. quality
    - When designing software architecture for a new product, it is often difficult to evaluate available design options and choose the optimal one. That often happens because it is unclear for the developers what criteria they should use to make design decisions and why.

Some developers rely on their previous engineering experience and personal preferences in practices, technologies, tools and patterns. The issue is that each dev team member has different preferences, opinions and assumptions. As a result, it may be difficult to reach consensus within a team and agree on some decisions. Arguing over subjective opinions and preferences may not only damage relationships between co-workers, but also won't necessarily lead to the software architecture optimised for achieving the business goals. What can developers do to make architecture design decisions more objective? One option would be to:

- start with investigating the non-functional requirements for the product,
- understand which software quality attributes it should be optimised for,
- then use that knowledge to choose the architecture options that allow their product to meet all the business requirements.

Before we move on, we first clarify what functional, non-functional requirements and quality attributes are. Functional requirements are those requirements that enlist what is functionally required from a system whereas Non-functional requirements are the criteria for evaluating *how* a software system should perform rather than *what* it should do. An example would be a requirement for a web API endpoint response time to be under 200ms. When we say that a software product should be "secure", "highly-available", "portable", "scalable" and so on, we are talking about its quality attributes. In other words, a software product must have certain quality attributes to meet certain non-functional requirements.

Topic- 006:

Teacher now defines quality control and quality assurance defining static and dynamic view of testing. Here product and process quality is defined saying that the quality assurance is fault prevention whereas quality control is fault detection.

In this lecture, the following topics are discussed:

- Quality control definition

  - A system of maintaining standards in manufactured products by testing a sample of the output against the specification or in other words it is the system of getting confidence that system is doing what it supposed to.
  - Business uses it to seek confidence that product quality is maintained or improved.
  - Quality control requires the business to create an environment in which both management and employees strive for perfection.

- Quality assurance definition
  - Quality assurance Is a way of preventing mistakes and defects in manufactured products and avoiding problems when delivering products or services to customers.
  - ISO 9000 defines quality assurance as "part of quality management focused on providing confidence that quality requirements will be fulfilled".
  - The confidence provided by quality assurance is twofold—internally to management and externally to customers, government agencies, regulators, certifiers, and third parties.
  - In general
    - Quality control is product centric whereas quality assurance is process centric.
    - It is fault/failure detection through static and/or dynamic testing of artefacts
    - It is examining of artefacts against pre-determined criteria to measure conformance
    - There is a difference of product quality and process quality
    - Quality Assurance is the process quality
    - It is fault prevention through process design and auditing
    - Creating processes, procedures, tools, jigs etc., to prevent faults from occurring
    - Prevent as much as possible defect injection
  - Process quality is one of a number of contributors to product quality.
  - Product quality is the overall quality of the product or in other words how well it conforms to the product requirements, specifications, and ultimately customer expectations.
  - Process quality focuses on how well some part of the process of manufacturing that product, and getting it into the customer’s hands, is working.
  - The process being analyzed in a particular case may have a very broad scope, or it may focus in on minute details of a single step. For example, the precise temperature used when molding a component, or the torque used when driving a specific screw.

- When a product quality issue is identified, good practice is to perform a root cause analysis to narrow down on the specific process steps that caused the issue, and then perform process experiments to develop and implement a corrective action.
- Quality Assurance and Quality Control are also processes in themselves that have quality parameters associated with their design and implementation.
- For example, how many parts from a lot do you decide to sample, and how do you select those parts, and which specific features do you inspect, and what do you do to react when a component quality issue is identified. These process design decisions can result in, or avoid, quality escapes to the ultimate product quality.

Lesson 03

Software Verification and Validation

Topic- 007:

Static testing involves manual or automated reviews of the documents. This review is done during an initial phase of testing to catch Defect early in STLC. It examines work documents and provides review comments.

- Informal Reviews: This is one of the types of review which doesn't follow any process to find errors in the document. Under this technique, you just review the document and give informal comments on it.
- Technical Reviews: A team consisting of your peers, review the technical specification of the software product and checks whether it is suitable for the project. They try to find any discrepancies in the specifications and standards followed. This review concentrates mainly on the technical documentation related to the software such as Test Strategy, Test plan and requirement specification documents.
- Walkthrough: The author of the work product explains the product to his team. Participants can ask questions if any. A meeting is led by the author. Scribe makes note of review comments
- Inspection: The main purpose is to find defects and meeting is led by a trained moderator. This review is a formal type of review where it follows a strict process to find the defects. Reviewers have a checklist to review the work products. They record the defect and inform the participants to rectify those errors.
- Static code Review: This is a systematic review of the software source code without executing the code. It checks the syntax of the code, coding standards, code optimization, etc. This is also termed as white box testing. This review can be done at any point during development.

Under Dynamic Testing, a code is executed. It checks for functional behavior of software system, memory/cpu usage and overall performance of the system. Hence the name "Dynamic". The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This testing is also called an Execution technique or validation testing.

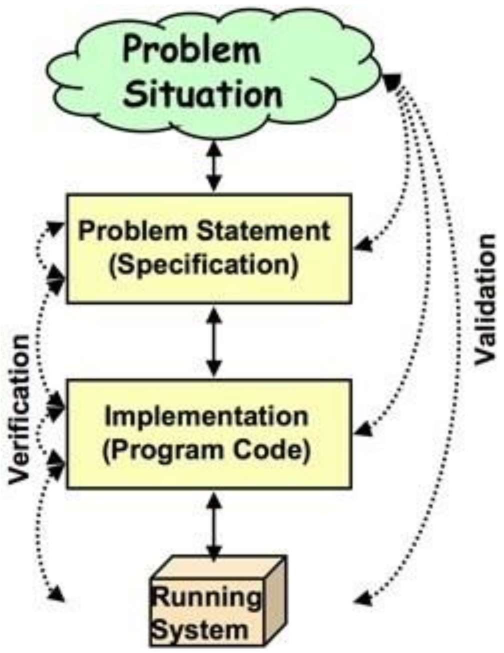
- Unit Testing: Under Unit Testing, individual units or modules are tested by the developers. It involves testing of source code by developers.
- Integration Testing: Individual modules are grouped together and tested by the developers. The purpose is to determine what modules are working as expected once they are integrated.
- System Testing: This type of testing is performed on the whole system by checking whether the system or application meets the requirement specification document.

Topic- 008:

Verification is process of assessing a software product or system during a particular phase to determine if it meets the requirements/conditions specified at beginning of that phase. Validation, on the other hand, is the process of testing a software product after its development process gets completed. It is conducted to determine whether a particular product meets the requirements that were specified for the product

In this lecture, the following topics are discussed:

- Verification and validation
    - Testing is a quality control measure used to verify that a product works as desired. Software testing provides a status report of the actual product in comparison to product requirements (written and implicit). Testing process must verify and validate whether the software fulfills conditions laid down for its release/use.
    - Priority Coverage: Exhaustive testing is impossible. We should perform tests efficiently and effectively, within budgetary and scheduling limitations. Therefore, testing needs to assign effort reasonably and prioritize thoroughly.
    - Balanced: Testing process must balance the written requirements, real-world technical limitations, and user expectations. The testing process and its results must be repeatable and independent of the tester, i.e., consistent, and unbiased.
    - Traceable: Documenting both the successes and failures helps in easing the process of testing. What was tested, and how it was tested, are needed as part of an ongoing testing process.
    - Deterministic: Problem detection should not be random in testing. We should know what we are doing, what are we targeting, what will be the possible outcome.
- Sometime in the 1990’s, I drafted a frequently asked question list for NASA’s IV&V facility. Here’s what I wrote on the meaning of the terms “validation” and “verification”:


Problem Situation
Problem Statement (Specification)
Implementation (Program Code)
Running System
Verification
Validation

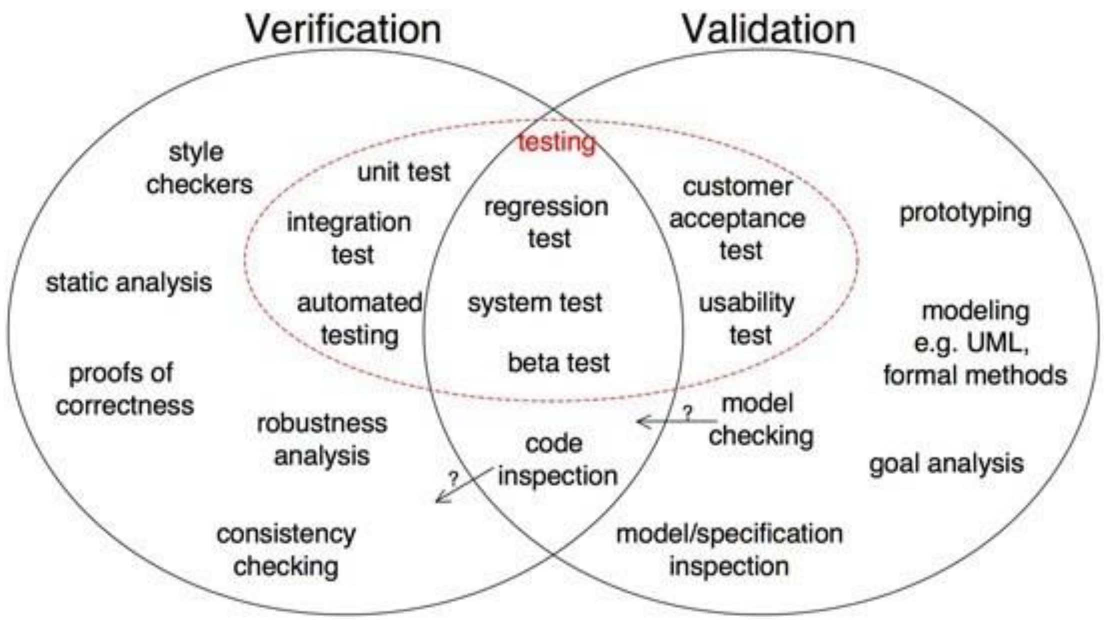
-

- The terms Verification and Validation are commonly used in software engineering to mean two different types of analysis. The usual definitions are:
- Validation: Are we building the right system?
- Verification: Are we building the system, right?

- Limitations of software testing include:
    - We cannot test a program completely
    - We can only test against system requirements
    - Exhaustive (total) testing is impossible in present scenario.
    - Time and likewise budget constraints normally require very careful planning of the testing effort.
    - Compromise between thoroughness and budget.
    - Test results used to make business decisions for release dates.
    - Even if you do find the last bug, you will never know it
    - You will run out of time before you run out of test cases
    - You cannot test every path
    - Every valid input cannot test
-
- Difference between validation and verification
- Validation is concerned with checking that the system will meet the customer's actual needs, while verification is concerned with whether the system is well-engineered, error-free, and so on. Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful. The distinction between the two terms is largely to do with the role of specifications. Validation is the process of checking whether the specification captures the customer's needs, while verification is the process of

checking that the software meets the specification. Verification includes all the activities associated with the producing high-quality software: testing, inspection, design analysis, specification analysis, and so on. It is a relatively objective process, in that if the various products and documents are expressed precisely enough, no subjective judgements should be needed in order to verify software.

- In contrast, validation is an extremely subjective process. It involves making subjective assessments of how well the (proposed) system addresses a real-world need. Validation includes activities such as requirements modelling, prototyping and user evaluation.
- In a traditional phased software lifecycle, verification is often taken to mean checking that the products of each phase satisfy the requirements of the previous phase. Validation is relegated to just the beginning and ending of the project: requirements analysis and acceptance testing. This view is common in many software engineering textbooks, and is misguided. It assumes that the customer's requirements can be captured completely at the start of a project, and that those requirements will not change while the software is being developed. In practice, the requirements change throughout a project, partly in reaction to the project itself: the development of new software makes new things possible. Therefore, both validation and verification are needed throughout the lifecycle.
- Finally, V&V is now regarded as a coherent discipline:" Software V&V is a system engineering discipline which evaluates the software in a systems context, relative to all system elements of hardware, users, and other software". (from Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards, by Dolores R. Wallace and Roger U. Fujii, NIST Special Publication 500-165) Having thus carefully distinguished the two terms, my advice to V&V practitioners was then to forget about the distinction, and think instead about V&V as a toolbox, which provides a wide range of tools for asking different kinds of questions about software. And to master the use of each tool and figure out when and how to use it. Here's one of my attempts to visualize the space of tools in the toolbox:



-
- A range of V&V techniques. Note that "modeling" and "model checking" refer to building and analyzing abstracted models of software behaviour, a very different kind of beast from scientific models used in the computational sciences
- For climate models, the definitions that focus on specifications don't make much sense, because there are no detailed specifications of climate models (nor can there be – they're built by iterative refinement like agile software development). But no matter – the toolbox approach still works; it just means some of the tools are applied a little differently. An appropriate toolbox for climate modeling looks a little different from my picture above, because some of these tools are more appropriate for real-time control systems, applications software, etc, and there are some missing from the above picture that are particular for simulation software. I'll draw a better picture when I've finished analyzing the data from my field studies of practices used at climate labs.
- Many different V&V tools are already in use at most climate modelling labs, but there is room for adding more tools to the toolbox, and for sharpening the existing tools (*what* and *how* are the subjects of my current research). But the question of how best to do this must proceed from a detailed analysis of current practices and how effective they are. There seem to be plenty of people wandering into this space, claiming that the models are insufficiently verified, validated, or both. And such people like to pontificate about what climate modelers ought to do differently. But anyone who pontificates in this way, but is unable to give a detailed account of which V&V techniques climate modellers currently use, is just blowing smoke. If you don't know what's in the toolbox already, then you can't really make constructive comments about what's missing.

Topic- 009:

Traditional definition of verification and validation leading to static and dynamic V&V is introduced in this lecture. The students are motivated why these two steps are required and how they are differentiated.

In this lecture, the following topics are discussed:

- Static V&V
- Dynamic V&V
- Critical system validation

Static Testing is a type of software testing in which software application is tested without code execution. Manual or automated reviews of code, requirement documents and document design are done in order to find the errors. The main objective of static testing is to improve the quality of software applications by finding errors in early stages of software development process.

Static testing involves manual or automated reviews of the documents. This review is done during an initial phase of testing to catch Defect early in STLC. It examines work documents and provides review comments. It is also called Non-execution testing or verification testing.

Examples of Work documents-

- Requirement specifications
- Design document
- Source Code
- Test Plans
- Test Cases
- Test Scripts
- Help or User document
- Web Page content

What is Dynamic Testing?

- Under **Dynamic Testing**, a code is executed. It checks for functional behavior of software system, memory/cpu usage and overall performance of the system. Hence the name "Dynamic"
- The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This testing is also called an Execution technique or validation testing.

- Dynamic testing executes the software and validates the output with the expected outcome. Dynamic testing is performed at all levels of testing and it can be either black or white box testing.

Main difference can be concluded as follows:

- Static testing was done without executing the program whereas Dynamic testing is done by executing the program.
- Static testing checks the code, requirement documents, and design documents to find errors whereas Dynamic testing checks the functional behavior of software system, memory/CPU usage and overall performance of the system.
- Static testing is about the prevention of defects whereas Dynamic testing is about finding and fixing the defects.
- Static testing does the verification process while Dynamic testing does the validation process.
- Static testing is performed before compilation whereas Dynamic testing is performed after compilation.
- Static testing techniques are structural and statement coverage while Dynamic testing techniques are Boundary Value Analysis & Equivalence Partitioning

Lesson 04

Software testing philosophies and testing strategies

Topic- 010:

Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability. This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data. Reliability measurement therefore requires a specially designed data set that replicates the pattern of inputs to be processed by the system. Safety assurance, on the other hand, and reliability measurement are quite different: Within the limits of measurement error, you know whether or not a required level of reliability has been achieved. However, quantitative measurement of safety is impossible. Safety assurance is concerned with establishing a confidence level in the system.

In this lecture, the following topics are discussed:

- Reliability Validation, Safety Assurance, Security assessment,

Topic- 011:

Software testing either involves code and is called white-box testing or it involves requirements conceived in the RS document where the relative techniques are called black-box testing techniques. Both have merits and demerits. Students are introduced white-box and black-box testing techniques and are made aware of their application scenarios.

In this lecture, the following topics are discussed:

- Testing philosophies in general:
  - Testing software is the reverse of the traditional scientific method, where you test the universe and then use the results of that experiment to refine your hypothesis. Instead, with software, if our “experiments” (tests) do not prove out our hypothesis (the assertions the test is making), we *change the system we are testing*. That is, if a test fails, it hopefully means that our software needs to change, not that our test needs to be changed. Sometimes we do also need to change our tests in order to properly reflect the current state of our software. it’s a natural part of this two-way scientific method–sometimes we’re learning that our tests are wrong, and sometimes our tests are telling us that our system is out of whack and needs to be repaired.
- White-box philosophies:
  - White box testing is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.

The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system. This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.

  - Example: A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

- Black box testing philosophies:
  - Black box testing is a software testing method in which the internal structure/design/implementation of the item being tested is **not** known to the tester. These tests can be functional or non-functional, though usually functional. This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories: Incorrect or missing functions, Interface errors, Errors in data structures or external database access, Behavior or performance errors, Initialization, and termination errors.
  - Example: A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser, providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.
- Glass-box testing philosophies:
  - Glass box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.
  - Glass Box Testing Coverage Techniques:
    - Statement Coverage - This technique aimed at exercising all programming statements with minimal tests.
    - Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once.
    - Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch is covered.
-

Topic- 012:

Software testing is an aligned activity with software development and it takes route of the underlying software development integration strategies. Students are taught how testing takes place considering various integration techniques.

In this lecture, the following topics are discussed:

- Testing strategies
- top down testing strategy
- bottom up testing strategy

Lesson 05

Analytical, model based and process-oriented strategies

Topic- 013:

Analytic testing strategy uses formal and informal techniques to access and prioritize risks that might occur during software testing. It takes a full overview of requirements, design and implementation of objects to determine the aim of testing. It collects complete information regarding software, target that is achieved and the data needed for testing the software. Model-based testing strategy tests the functionality of the software. It identifies the domain of data and selects suitable test cases as per the probability of errors in that domain. Methodical testing strategy tests the function and status of software according to checklist, based on the user requirements. This strategy is used to test the functionality, reliability, usability and performance of the software. Process-oriented testing strategy tests the software from the existing standards i.e IEEE standards. It ensures the functionality of the software by using automated testing tools. Dynamic testing strategy tests the software after having common decision of the testing team. This strategy gives information regarding the software, for e.g., the test cases used for testing the errors present in it. Philosophical testing strategy tests the software by assuming that any component of the software terminates the functioning anytime.

Topic- 014:

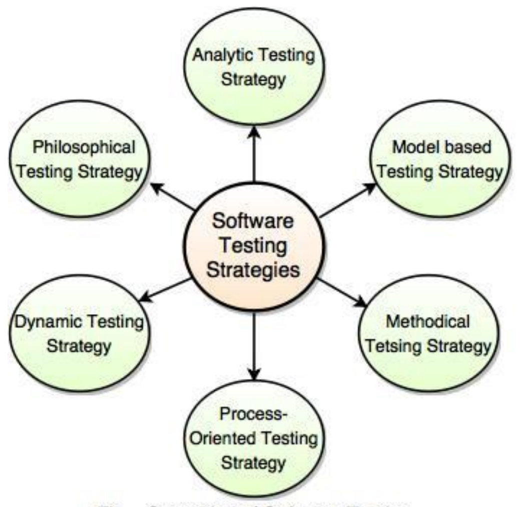
There is a statement given in lecture 1 saying “Exhaustive testing is not practical even for the simplest programs”. Here, students are taught why exhaustive testing is not possible and then they are given an understanding regarding testing comprehensiveness. This is a precursor to definition of definitions of “test requirement”, “test coverage” and leads to our definition of testing process.

In this lecture, the following topics are discussed:

- Analytical, model based, process-oriented strategies: In the success of the test effort, accuracy of the test plans and estimation of test strategy are the most powerful factors. The testers select different types of software testing strategies depending on the nature and size of the software. Following are the commonly used software testing strategies:

  i) Analytic testing strategy

  - Analytic testing strategy uses formal and informal techniques to access and prioritize risks that might occur during software testing.
  - It takes a full overview of requirements, design and implementation of objects to determine the aim of testing.
  - It collects complete information regarding software, target that is achieved and the data needed for testing the software.


Analytic Testing Strategy
Philosophical Testing Strategy
Model based Testing Strategy
Software Testing Strategies
Dynamic Testing Strategy
Methodical Tetsing Strategy
Process-Oriented Testing Strategy
Fig. - Strategies of Software Testing

ii) Model-based testing strategy

- Model-based testing strategy tests the functionality of the software.
- It identifies the domain of data and selects suitable test cases as per the probability of errors in that domain.

iii) Methodical testing strategy

- Methodical testing strategy tests the function and status of software according to checklist, based on the user requirements.
- This strategy is used to test the functionality, reliability, usability and performance of the software.

iv) Process-oriented testing strategy

- Process-oriented testing strategy tests the software from the existing standards i.e IEEE standards.
- It ensures the functionality of the software by using automated testing tools.

v) Dynamic testing strategy

- Dynamic testing strategy tests the software after having common decision of the testing team.
- This strategy gives information regarding the software, for e.g., the test cases used for testing the errors present in it.

vi) Philosophical testing strategy

- This strategy tests the software by assuming that any component of the software terminates the functioning anytime.
- For testing the software, it takes help from software developers, users and system analysis.

- Testing Comprehensiveness

  How well does your developed software or application behave is something that can make or break your business? The behaviour of your application depends on how comprehensively it is tested. Compromising in testing means compromising on your customer count and your business reputation. So, testing your application should never be overlooked. To achieve comprehensive software testing, there are certain aspects that shouldn't be overlooked. Considering these aspects can prove beneficial for the businesses as well as customers before the release of the application.

- Achieve maximum test coverage by covering all functional points
  Testing an application completely means achieving maximum test coverage by covering all the functional points. Due to the tight deadlines, it becomes difficult for organizations to test the software completely. Sometimes, organizations might go for a release without achieving maximum test coverage, and this results in the failure of application putting down the business and brand down. So, an application should be tested covering all the functional points to make sure that the application works well when end users are using it.
- Adopting both manual testing and automation testing
  The compelling need of achieving "faster time to market" and the advancements in software testing made automation testing to stand at the top. But, ignoring manual testing completely will not yield better outcome, the combination of both automation and manual testing enables to deliver quality software release.

- Automation is the best alternative to save time and effort on doing certain tasks which are tedious to do manually. So, we can say that automation cannot be a complete replacement for manual testing. At first, the software should be tested manually and when it gains some kind of stability, automation can be implemented accordingly.
- Exploratory Testing to increase the application quality
  Exploratory Testing is a process of finding defects in an application without test cases, it is a process of executing ad hoc tests developed logically. At most organizations, this is an underrated process but it is actually a powerful way to increase the application quality before the release.
- Occasional defects can impact end user experience
  When testing an application, testers will report a few defects that are occasional, meaning they don't reproduce. So, it becomes in communicating with developers to fix such defects and it intends to ignore them. But when the application goes live and if end users come across same the defect, then it will impact the end user experience. Therefore, occasional defects shouldn't be ignored, they should be recorded as a video and reported.

Lesson 06

Testing phases

Topic- 015:

During this first round of testing, the program is submitted to assessments that focus on specific units or components of the software to determine whether each one is fully functional. The main aim of this endeavor is to determine whether the application functions as designed. In this phase, a unit can refer to a function, individual program or even a procedure, and a White-box Testing method is usually used to get the job done. One of the biggest benefits of this testing phase is that it can be run every time a piece of code is changed, allowing issues to be resolved as quickly as possible. It's quite common for software developers to perform unit tests before delivering software to testers for formal testing.

In this lecture, the following topics are discussed:

- Testing phases
    - Unit Testing: Unit testing is the first stage of software testing levels. During this stage, testers evaluate individual components of the system to see if these components are functioning properly on their own.
    - Integration Testing: Testers perform integration testing in the next phase of testing. Here, they test individual components of the system and then test them as a collective group. This allows software testers to determine the performance of individual components as a group and identify any problems in the interface between the modules and functions.
    - System Testing: System testing is the final stage of the verification process. In this stage, testers see whether or not the collective group of integrated components is performing optimally. The process is crucial for the quality life cycle, and testers strive to evaluate if the system can fulfill the quality standards and complies with all major requirements.
    - Acceptance Testing: Acceptance testing is the final stage of the QA test cycle. It helps evaluate if the application is ready to be released for user consumption. Typically, testers carry out this phase with the help of the representatives of the customer who test the application by using it. Therefore, they will check if the application can perform all the functions specified.

- unit testing
    - During this first round of testing, the program is submitted to assessments that focus on specific units or components of the software to determine whether each one is fully functional. The main aim of this endeavor is to determine whether the application functions as designed. In this phase, a unit can refer to a function,

individual program or even a procedure, and a White-box Testing method is usually used to get the job done. One of the biggest benefits of this testing phase is that it can be run every time a piece of code is changed, allowing issues to be resolved as quickly as possible. It's quite common for software developers to perform unit tests before delivering software to testers for formal testing.

-

Topic- 016:

Integration testing allows individuals the opportunity to combine all of the units within a program and test them as a group. This testing level is designed to find interface defects between the modules/functions. This is particularly beneficial because it determines how efficiently the units are running together. Keep in mind that no matter how efficiently each unit is running, if they aren't properly integrated, it will affect the functionality of the software program. In order to run these types of tests, individuals can make use of various testing methods, but the specific method that will be used to get the job done will depend greatly on the way in which the units are defined.

System testing is the first level in which the complete application is tested as a whole. The goal at this level is to evaluate whether the system has complied with all of the outlined requirements and to see that it meets Quality Standards. System testing is undertaken by independent testers who haven't played a role in developing the program. This testing is performed in an environment that closely mirrors production. System Testing is very important because it verifies that the application meets the technical, functional, and business requirements that were set by the customer.

In this lecture, the following topics are discussed:

- integration testing
- system testing

Topic- 017:

The final level, Acceptance testing (or User Acceptance Testing), is conducted to determine whether the system is ready for release. During the Software development life cycle, requirements changes can sometimes be misinterpreted in a fashion that does not meet the intended needs of the users. During this final phase, the user will test the system to find out whether the application meets their business' needs. Once this process has been completed and the software has passed, the program will then be delivered to production.

In this lecture, the following topics are discussed:

- acceptance testing
- alpha and beta testing

Topic- 018:

Students need to understand why we need to do unit, integration and subsystem / system testing without avoiding any of these. The students understand how various aspects of system validation are handled at different steps,

In this lecture, the following topics are discussed:

- Focus of unit, integration and (sub-)system
- Artifacts involved in testing
- The big picture

Lesson 07

Structural testing techniques

Topic- 019:

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation. The other names of structural testing include clear box testing, open box testing, logic driven testing or path driven testing. Advantages include forcing test developer to reason carefully about implementation, revealing errors in "hidden" code, and spotting dead code or other issues with respect to best programming practices. However, disadvantages include that these techniques are expensive as one has to spend both time and money to perform white box testing, every possibility that few lines of code is missed accidentally and in-depth knowledge about the programming language is necessary to perform white box testing.

In this lecture, the following topics are discussed:

- Structural testing techniques
  - Structural testing carried out to test the structure of code. It is also known as White Box testing or Glass Box testing.
- Structural Testing coverage techniques
  - Statement Coverage
    - It aims to test all the statements present in the program. Adequacy Criterion should be equal to 1 to ensure 100% coverage. It is a good measure of testing each part in terms of statements, but it is not a good technique for testing the control flow.
  - Branch Coverage
    - It is also known as Decision coverage testing. Branch coverage aims to test all the branches or edges at least once in the test suite or to test each branch from a decision point at least once. It provides solution for the problem faced in Statement coverage. Branch Testing provides a better coverage than Statement testing but it too has its shortcomings. It does not provide a good coverage from different conditions that lead from node 1 to node 2 as it covers that branch only once.
  - Condition Coverage
    - It aims to test individual conditions with possible different combination of Boolean input for the expression. It is a modification of Decision coverage. But it provides better coverage. However, when compound conditions are involved, the number of test cases may increase exponentially. Which is not favorable.
  - Path Coverage

        - Path coverage aims to test the different path from entry to the exit of the program. Which is a combination of different decisions taken in the sequence.
        - For example, a loop can go on and on. Which help in finding out the redundant test cases. So, cyclomatic complexity helps aiming to test all linearly independent paths in a program at least once.
-
- Test Selection and Adequacy Criteria:
    - Test selection:
        - Several items must be considered when determining the approaches to the organization of the testing, the selection of the tests, and the methods to use when conducting the tests. Any approach must be compatible with the: customer’s requirements, development strategies, testing objectives, system environment, mandated methodologies.
    - Adequacy criteria:
        - The criteria can be viewed as representing minimal standards for testing a program. The application scope of adequacy criteria also includes: helping testers to select properties of a program to focus on during test, helping testers to select a test data set for a program based on the selected properties, supporting testers with the development of quantitative objectives for testing, indicating to testers whether or not testing can be stopped for that program.
        - A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion. Using the selected adequacy criterion, a tester can terminate testing when he/she has exercised the target structures, and have some confidence that the software will function in manner acceptable to the user.
-

Topic- 020:

A graph G = (V, E) consists of a (finite) set denoted by V, or by V(G) if one wishes to make clear which graph is under consideration, and a collection E, or E(G), of unordered pairs {u, v} of distinct elements from V. Each element of V is called a vertex or a point or a node, and each element of E is called an edge or a line or a link. Formally, a graph G is an ordered pair of disjoint sets (V, E), where E Í V × V. Set V is called the vertex or node set, while set E is the edge set of graph G. Typically, it is assumed that self-loops (i.e. edges of the form (u, u), for some u Î V) are not contained in a graph. A graph G = (V, E) is directed if the edge set is composed of ordered vertex (node) pairs. A graph is undirected if the edge set is composed of unordered vertex pair.

In this topic, students are introduced with based knowledge regarding graphs, traversing graphs and constructing them, the following topics are discussed:

- Parts of a graph: nodes, edges, cycles

Lesson 08

Control-flow graph-based testing techniques

Topic- 021:

Graph Based Testing is also called as State Based Testing. It provides a framework for model-based testing as well. Black-box methods are based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph. Considering transaction flow testing where nodes represent steps in some transaction and links represent logical connections between steps that need to be validated.

Our agenda is to consider, however, white-box testing and conduct code-based unit testing constructing control-flow graph. We introduce the following steps:

- Build a graph model.
- Identify the test requirements.
- Select test paths to cover those requirements.

In order to design test design cases following steps are used:

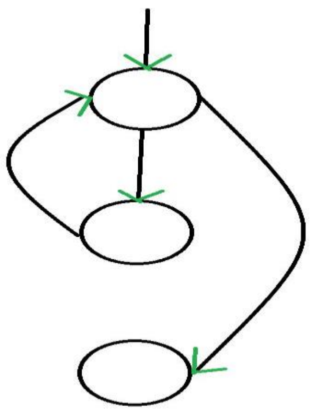
- Understanding the system
- Identifying states, inputs and guards.
- Create a state graph model of the application.
- Verify whether State graph that we modeled is correct in all details.
- Generate sequence of test actions.

In this lecture, the following topics are discussed:

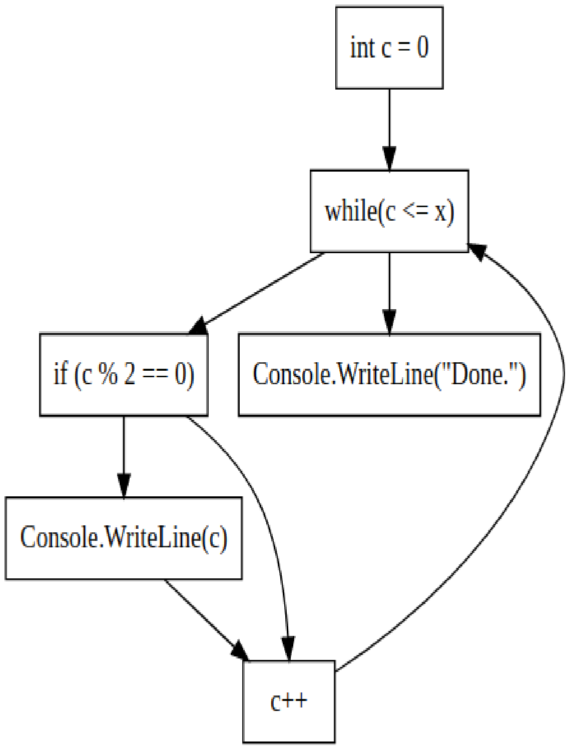
- Control-flow based testing
  - Control flow testing is a type of software testing that uses program's control flow as a model. Control flow testing is a structural testing strategy. This testing technique comes under white box testing. For the type of control flow testing, all the structure, design, code, and implementation of the software should be known to the testing team.
  - Developers often use this type of testing method to test their own code and own implementation as the design, code and the implementation is better known to the developers. This testing method is implemented with the intention to test the logic of the code so that the user requirements can be fulfilled. Its main application is to relate the small programs and segments of the larger programs.
- Control flow graph (CFG):
  - A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications. Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit. Control flow graph is

process oriented. Control flow graph shows all the paths that can traversed during a program execution. Control flow graph is a directed graph. Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks.

- Example of a CFG:
    - while:


while

    - Control-flow graph of SUT:



- Control flow graph (CFG)
- Example of a CFG

Topic- 022:

This topic covers important aspect of analysis of control-flow graph to extract paths and to devise test cases from selected paths. Idea of sufficiency of test cases reported as test requirement and associated coverage criteria is discussed with students.

In this lecture, the following topics are discussed:

- From CFG to path selection
- From paths to test cases
- Coverage criteria for CFG based testing

Lesson 09

Dataflow based software testing

Topic- 023:

Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used. Dataflow testing helps address issues such as case when a variable that is declared but never used within the program, a variable that is used but never declared, a variable that is defined multiple times before it is used and deallocating a variable before it is used.

In this lecture, the following topics are discussed:

- Dataflow Testing
    - Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used. Graph Coverage Criteria is given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in path(T) that meets the test requirement tr.
    - Two types:
        - Structural coverage criteria: Define a graph just in terms of nodes and edges.
        - Data flow coverage criteria: Requires a graph to be annotated with references to variables.

- Dataflow Testing – anomalies
    - In Software testing, Anomaly refers to a result that is different from the expected one. This behaviour can result from a document or also from a testers notion and experiences.
    - An Anomaly can also refer to a usability problem as the testware may behave as per the specification, but it can still improve on usability. Sometimes, the anomaly can also referred as a defect / Bug.
    - Data Flow Anomalies are identified while performing while box testing or Static Testing. Data flow anomalies are represented using two characters based on the sequence of actions. They are defined (d), killed (k), and used (u). There are nine possible combinations based on these 3 sequences of actions which are dd, dk, du, kd, kk, ku, ud, uk, uu. The below table clearly shows which one of these combinations are accepted and which one of these are suspected to be an anomaly.

Combination	Description	Anomaly possibilities
dd	Defined the data objects twice	Harmless but suspicious
dk	Defined the data object but killed it without using it.	Bad Programming Practice
du	Defined the data object and using it	NOT an Anomaly
kd	Killed the Data Object and redefined	NOT an Anomaly
kk	Killed the Data Object and killed it again	Bad Programming Practice
ku	Killed the Data Object and then used	Defect
ud	Used the Data Object and redefined	NOT an Anomaly
uk	Used the Data Object and Killed	NOT an Anomaly
uu	Used the Data Object and used it again	NOT an Anomaly

Topic- 024:

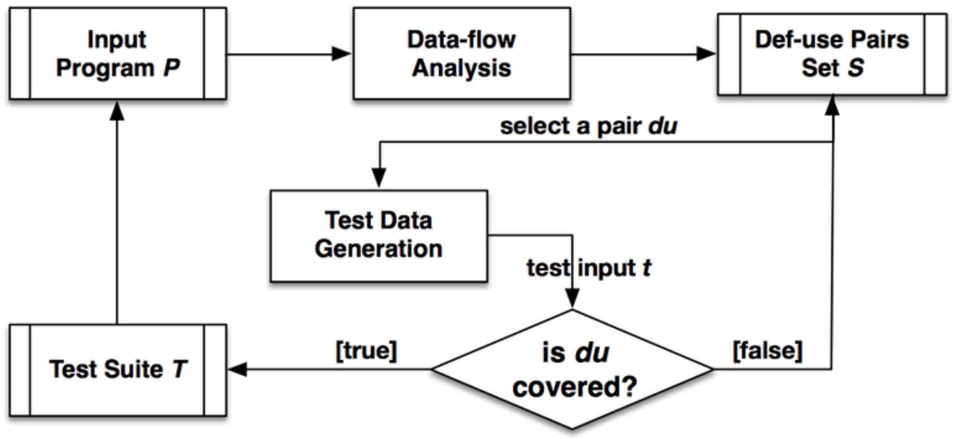
Dataflow based coverage criteria consider dataflow annotations on control flow graphs and coverage criteria are based on def-use pairs, all-def, all-uses and their combinations. We also discuss how various criteria subsume other criteria and explain to the students why some criteria require more test cases.

In this lecture, the following topics are discussed:

- Dataflow Testing Coverage Criteria

Data Flow Testing is a specific strategy of software testing that focuses on data variables and their values. It makes use of the control flow graph. When it comes to categorization Data flow testing will can be considered as a type of white box testing and structural types of testing. It keeps a check at the data receiving points by the variables and its usage points. It is done to cover the path testing and branch testing gap.

The process is conducted to detect the bugs because of the incorrect usage of data variables or data values. For e.g. Initialization of data variables in programming code, etc.



What is Data flow Testing?

- The programmer can perform numerous tests on data values and variables. This type of testing is referred to as data flow testing.
- It is performed at two abstract levels: static data flow testing and dynamic data flow testing.
- The static data flow testing process involves analyzing the source code without executing it.
- Static data flow testing exposes possible defects known as data flow anomaly.
- Dynamic data flow identifies program paths from source code.

Let us understand this with the help of an example.

```
1. read x;
2. If(x>0)                  (1, (2, t), x), (1, (2, f), x)
3. a= x+1                   (1, 3, x)
4. if (x<=0) {              (1, (4, t), x), (1, (4, f), x)
5. if (x<1)                 (1, (5, t), x), (1, (5, f), x)
6. x=x+1; (go to 5)         (1, 6, x)
else
7. a=x+1                    (1, 7, x)
8. print a;                 (6,(5, f)x), (6,(5,t)x)
                            (6, 6, x)
                            (3, 8, a), (7, 8, a).
```

There are 8 statements in this code. In this code we cannot cover all 8 statements in a single path as if 2 is valid then **4, 5, 6, 7** are not traversed, and if 4 is valid then statement 2 and 3 will not be traversed.

Hence we will consider two paths so that we can cover all the statements.

x= 1

Path – 1, 2, 3, 8

Output = 2

If we consider x = 1, in step 1; x is assigned a value of 1 then we move to step 2 (since, x>0 we will move to statement 3 (a= x+1) and at end, it will go to statement 8 and print x =2.

For the second path, we assign x as 1

Set x= -1

Path = 1, 2, 4, 5, 6, 5, 6, 5, 7, 8

Output = 2

x is set as 1 then it goes to step 1 to assign x as 1 and then moves to step 2 which is false as x is smaller than 0 (x>0 and here x=-1). It will then move to step 3 and then jump to step 4; as 4 is true (x<=0 and their x is less than 0) it will jump on 5 (x<1) which is true and it will move to step 6 **(x=x+1)** and here x is increased by 1.

So,

x=-1+1

x=0

x become 0 and it goes to step 5(x<1),as it is true it will jump to step

6 (x=x+1)

x=x+1

x= 0+1

x=1

x is now 1 and jump to step 5 (x<1) and now the condition is false and it will jump to step 7 **(a=x+1)** and set a=2 as x is 1. At the end the value of a is 2. And on step 8 we get the output as 2.

Steps of Data Flow Testing

creation of a data flow graph.

- Selecting the testing criteria.
- Classifying paths that satisfy the selection criteria in the data flow graph.
- Develop path predicate expressions to derive test input.
- The life cycle of data in programming code

Definition: it includes defining, creation and initialization of data variables and the allocation of the memory to its data object.

- Usage: It refers to the user of the data variable in the code. Data can be used in two types as a predicate(P) or in the computational form(C).
- Deletion: Deletion of the Memory allocated to the variables.

Types of Data Flow Testing

- Static Data Flow Testing

  No actual execution of the code is carried out in Static Data Flow testing. Generally, the definition, usage and kill pattern of the data variables is scrutinized through a control flow graph.

- Dynamic Data Flow Testing: The code is executed to observe the transitional results. Dynamic data flow testing includes:
  - Identification of definition and usage of data variables.
  - Identifying viable paths between definition and usage pairs of data variables.
  - Designing & crafting test cases for these paths.

Advantages of Data Flow Testing

- Variables used but never defined,
- Variables defined but never used,
- Variables defined multiple times before actually used,
- DE allocating variables before using.

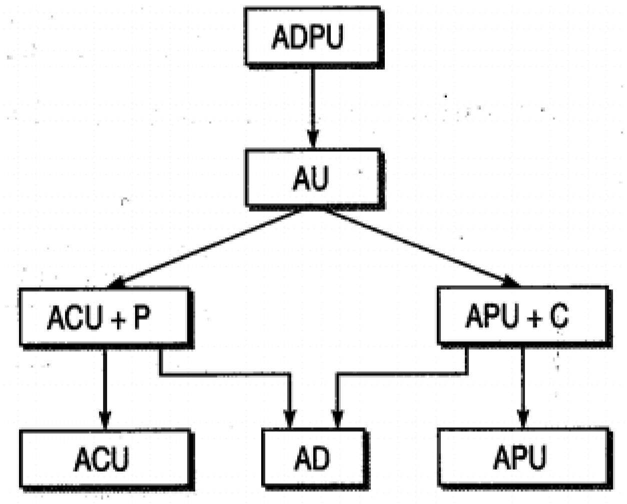
Data Flow Testing Limitations

- Testers require good knowledge of programming.
- Time-consuming
- Costly process.

Data Flow Testing Coverage

- All definition coverage: Covers "sub-paths" from each definition to some of their respective use.
- All definition-C use coverage: "sub-paths" from each definition to all their respective C use.
- All definition-P use coverage: "sub-paths" from each definition to all their respective P use.
- All use coverage: Coverage of "sub-paths" from each definition to every respective use irrespective of types.
- All definition use coverage: Coverage of "simple sub-paths" from each definition to every respective use.

Data Flow Testing Strategies



Following are the test selection criteria

1. All-defs: For every variable x and node i in a way that x has a global declaration in node I, pick a comprehensive path including the def-clear path from node i to

- Edge (j,k) having a p-use of x or
- Node j having a global c-use of x

2. All c-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to all nodes j having a global c-use of x in j.

3. All p-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to all edges (j,k) having p-use of x on edge (j,k).

4. All p-uses/Some c-uses: it is similar to all p-uses criterion except when variable x has no global p-use, it reduces to some c-uses criterion as given below

5. Some c-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to some nodes j having a global c-use of x in node j.

6. All c-uses/Some p-uses:it is similar to all c-uses criterion except when variable x has no global c-use, it reduces to some p-uses criterion as given below:

7. Some p-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including def-clear paths from node i to some edges (j,k) having a p-use of x on edge (j,k).

8. All uses:it is a combination of all p-uses criterion and all c-uses criterion.

9. All du-paths:For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including all du-paths from node i

- To all nodes j having a global c-use of x in j and
- To all edges (j,k) having a p-use of x on (j,k).

Data Flow Testing Applications

As per studies defects identified by executing **90%** "data coverage" is twice as compared to bugs detected by **90%** branch coverage.

The process flow testing is found effective, even when it is not supported by automation.

It requires extra record keeping; tracking the variables status. The computers help easy tracking of these variables and hence reducing the testing efforts considerably. Data flow testing tools can also be integrated into compilers.

Conclusion

Data is a very important part of software engineering. The testing performed on data and variables play an important role in software engineering. Hence this is a very important part and should be properly carried out to ensure the best working of your product.

- Subsumption of coverage criteria
    - o Criteria subsumption : A test criterion C1 subsumes C2 if and only if every set of test cases that satisfies criterion C1 also satisfies C2

- Must be true for every set of test cases
- Example: color criteria for the jelly bean: {yellow, green, orange, white}
- If we satisfy flavor criteria, we'll satisfy color criteria
- Example : If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement
- Coverage does not depend on the number of test cases
- T0 , T1 : T1 >coverage T0   T1 <cardinality T0
- T1 , T2 : T2 =coverage T1   T2 >cardinality T1
- Small test cases make failure diagnosis easier
- A failing test case in T_2 gives more information for fault localization than a failing test case in T_1

Lesson 10

Program slicing based software testing

Topic- 025:

Program slicing works by finding the parts of a program relevant to the value of a chosen set of variables at some chosen point in a program. A slice is constructed by deleting the parts of the program that are irrelevant to those values.

The point of interest is usually identified by annotating the program with line numbers which identify each primitive statement and each branch point. The term 'slicing criterion' is used for the point of interest together with the set of variables whose value the slice must preserve. In this article the point of interest will be indicated by adding a comment to the program. In general slices are constructed for a set of variables, but herein only slices constructed for a single variable will be considered. For a slicing criterion consisting of a variable v and a point of interest n, the slice is constructed for v at n.

Having picked a slicing criterion one of two forms of slice can be constructed: a backward slice or a forward slice. The former contains the statements of the program which can have some effect on the slicing criterion, whereas a forward slice contains those statements of the program which are affected by the slicing criterion. Backward slices can assist a developer by helping to locate the parts of the program which contain a bug. Forward slicing can be used to predict the parts of a program that will be affected by a modification.

In this lecture, the following topics are discussed:

- Program Slice based Testing

Topic- 026:

Slicing was originally proposed by Mark Weiser in his seminal PhD thesis in 1979. Since then a great deal of research work involving the development, improvement, and extension of the algorithms for constructing program slices has been conducted. Until recently the only tools for constructing slices were academic play things which catered only for unrealistically small subsets of programming languages, but 1996 witnessed the production of a publicly available slicing tool for a real programming language. It was developed by Jim Lyle, Dolores Wallace, James Graham, Keith Gallagher, Joseph Poole, and David Binkley for the American National Institute of Standards and Technology. The Unravel team has, philanthropically, made its tool and its source code freely available on the Web.

We make use of examples and explain various uses of program slicing

In this lecture, the following topics are discussed:

- Program Slice – examples
    - Program slice is a decomposition technique that extracts statements relevant to a particular computation from a program
    - Program slicing describes a mechanism which allows the automatic generation of a slice.
    - **Slicing criterion <s , v>**
        - Where s specifies a location (statement s) and v specifies a variable (v)
        - All statements affecting or affected by the variables mentioned in the slicing criterion becomes a part of the slice

Original program:

```
1 begin
2 read(x,y)
3 total := 0.0
4 sum := 0.0
5 if x <= 1
      6   then sum := y
      7   else begin
      8         read(z)
      9         total := x*y
      10        end
      11   write(total, sum)
      12   end.
```

Slice criterion:

<12, z>

```
        begin
        read(x,y)
    if x <= 1
            then
            else read(z)
        end.
```

Slice criterion:

<9, x>

```
        begin
        read(x,y)
        end.
```

Slice criterion:

<12, total>

```
         begin
        read(x,y)
        total := 0
        if x <= 1
            then
          else total := x*y
        end.
```

1. **Program Slice – Uses in testing:**
   Slicing or program slicing is a technique used in software testing which takes a slice or a group of program statements in the program for testing particular test conditions or cases and that may affect a value at a particular point of interest.
   - A static slice of a program contains all statements that may affect the value of a variable at any point for any arbitrary execution of the program.
   - A dynamic slice of a program contains all the statements that actually affect the value of a variable at any point for a particular execution of the program.

```
int z = 10;
int n;
cin >> n;
```

```
int sum = 0;
if (n > 10)
    sum = sum + n;
else
    sum = sum - n;
cout << "Hey";
```

Static slice for the variable **sum**:

```
int n;
cin >> n;
int sum = 0;
if (n > 10)
    sum = sum + n;
else
    sum = sum - n;
```

Dynamic slice for the variable **sum** when n = 22;

```
int n;
cin >> n;
int sum = 0;
if (n > 10)
    sum = sum + n;
```

- Program Slice – Uses in testing

Lesson 11

Control-flow testing – worked example

Topic- 027:

Considering introduction to control-flow graph based technique lesson, we now explain how do we make use of IDE such as visual studio to do unit testing. We explain the example code and prepare students for lab on unit testing.

In this lecture, the following topics are discussed:

- Testing based on control-flow graph – Lab
- Preparation for Lab on unit testing
- Introduction to System Under Test (SUT)

Topic- 028:

Once the students have had a look at the experiment code, they are made to develop control-flow of the SUT. This leads them to develop test cases manually. Once this step is over, they are ready to use M/S Visual Studio IDE and execute their unit test cases.

In this lecture, the following topics are discussed:

- Control-flow graph of SUT: Test design consists of following stages:
    - Test strategy
    - Test planning
    - Test specification
    - Test procedure
- The purpose of the first test case in any unit test specification should be to execute the unit under test in the simplest way possible
- Test cases should be designed to show that the unit under test does what it is supposed to do.
- Test cases should be enhanced, and further test cases should be designed to show that the software does not do that it is not supposed to do.
- Where appropriate, test cases should be designed to address issues related to performance, safety and security requirements.
- Add more test cases to the unit test specification to achieve specific test coverage objectives.

- Where coverage objectives are not achieved, analysis must be conducted to determine why. Failure to achieve a coverage objective may be due to: Infeasible paths or conditions, unreachable or redundant code, Insufficient test cases

- Unit testing with Microsoft Visual Studio:
  - Use Visual Studio to define and run unit tests to maintain code health, ensure code coverage, and find errors and faults before your customers do. Run your unit tests frequently to make sure your code is working properly.
  - Create test:
    1- Open the project that you want to test in Visual Studio. For the purposes of demonstrating an example unit test, this article tests a simple "Hello World" project named HelloWorldCore.
    2- In **Solution Explorer**, select the solution node. Then, from the top menu bar, select **File** > **Add** > **New Project**.
    3- In the new project dialog box, find a unit test project template for the test framework you want to use and select it.
    4- Click **Next**, choose a name for the test project, and then click **Create**.
    5- In the unit test project, add a reference to the project you want to test by right-clicking on **References** or **Dependencies** and then choosing **Add Reference**.
    6- Select the project that contains the code you'll test and click **OK**.
    7- Add code to the unit test method.
  - Run test:

    1. Open Test Explorer. To open Test Explorer, choose **Test** > **Test Explorer** from the top menu bar.
    2. Run your unit tests by clicking **Run All**.

- Coverage with Microsoft Visual Studio:
  - To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise or 'cover' a large proportion of your code.
    1- On the Test menu, select Analyze Code Coverage for All Tests.
    2- After the tests have run, to see which lines have been run, choose Show Code Coverage Coloring in the Code Coverage Results window. By default, code that is covered by tests is highlighted in light blue.
    3- If the results show low coverage, investigate which parts of the code are not being exercised, and write more tests to cover them. Development teams typically aim for about 80% code coverage. In some situations, lower coverage is acceptable. For example, lower coverage is acceptable where some code is generated from a standard template

-

-
- Unit testing – test case design
-

Topic- 029:

The students are made aware of the use of M/S Visual Studio

In this lecture, the following topics are discussed:

- Unit testing with Microsoft Visual Studio
- Coverage with Microsoft Visual Studio

Lesson 12

Integration Testing

Topic- 030:

Integration testing is defined as a type of testing where software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated, Although each software module is unit tested, defects still exist for various reasons like

- A Module, in general, is designed by an individual software developer whose understanding and programming logic may differ from other programmers. Integration Testing becomes necessary to verify the software modules work in unity
- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling could cause issues.

Students understand the relevance of integration testing in this topic.

Topic- 031:

Students are explained the nature of test case for integration testing. An example that can be given to students is as follows:

T ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mailbox select the email and click a delete button	Selected email should appear in the Deleted/Trash folder

Teacher first explains variety of strategies to execute Integration testing such as big bang approach, incremental approach: which is further divided into top down approach, bottom up approach and sandwich approach which is combination of top down and bottom up. Students are then explained functional decomposition-based integration and associated testing strategy. Once done, students are taught call-graph based and path-based integration testing strategies as well.

In this lecture, the following topics are discussed:

- Functional decomposition-based Integration
- Call Graph based Integration
- Path based Integration

MM path extraction for integration testing is an important method used for white-box integration testing. Method/message path (MM path) is defined as an interleaved sequence of method executions linked by messages. It presents well the interactions between the methods of object-oriented software, and hence fits for object-oriented integration testing. MM paths are independently testable units (ITF) and therefore they represent uses of a system. This type of testing also plays an important role in identifying if there is any mismatch in design of a system and corresponding implementation.

Lesson 13

System testing

Topic- 032:

System testing is a form of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system. System test falls under the black box testing category of software testing.

As with almost any software engineering process, software testing has a prescribed order in which things should be done. Unit testing performed on each module or block of code during development. It is normally done by the programmer who writes the code. Integration testing done before, during and after integration of a new module into the main software package. This involves testing of each individual code module. One piece of software can contain several modules which are often created by several different programmers. It is crucial to test each module's effect on the entire program model. System testing, which is the topic under discussed, is done by a professional testing agent on the completed software product before it is introduced to the market. Acceptance testing - beta testing of the product done by the actual end users. Here we introduce students with system testing and its aspects.

In this lecture, the following topics are discussed:

- System testing
- System testing aspects

Topic- 033:

Considering a system under test (SUT), there are several system level testing types considering functional and non-functional aspects of a system. At the system level, we extract test cases considering various SRS components such as use-cases, use-case descriptions, user stories, etc.

For the non-functional aspects, we have testing techniques pertaining to these non-functional aspects. We consider a couple of examples. Usability testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives. Load testing is necessary to know that a software solution will perform under real-life loads. Regression testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time. Recovery testing - is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes. Migration testing- is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

In this lecture, the following topics are discussed:

- System testing – Non-functional aspects
- System testing – Functional aspects

Lesson 14

Use cases for testing

Topic- 034:

A use case is a tool for defining the required user interaction. Business experts and developers must have a mutual understanding about the requirement, as it's very difficult to attain. Any standard method for structuring the communication between them will really be a boon. It will, in turn, reduce the miscommunications and here is the place where Use case comes into the picture.
Use case plays a significant role in the distinct phases of the Software Development Life Cycle. Use Case depends on 'User Actions' and 'Response of System' to the User Actions. It is the documentation of the 'Actions' performed by the Actor/User and the corresponding 'Behaviour' of the System to the User 'Actions'. Use Cases may or may not result in achieving a goal by the 'Actor/User' on interactions with the system.

Sunny day Use Cases are the primary cases that are most likely to happen when everything does well. These are given high priority than the other cases. Once we have completed the cases, we give it to the project team for review and ensure that we have covered all the required cases. Rainy day Use Cases, on the other hand, can be defined as the list of edge cases. The priority of such cases will come after the 'Sunny Use Cases'.

We can seek the help of Stakeholders and product managers to prioritize the cases. Use case descriptions include brief description (A brief description explaining the case), actor (Users that are involved in Use Cases Actions), precondition (Conditions to be Satisfied before the case begins), basic flow('Basic Flow' or 'Main Scenario' is the normal workflow in the system. It is the flow of transactions done by the Actors on accomplishing their goals.

When the actors interact with the system, as it's the normal workflow, there won't be any error and the Actors will get the expected output), alternate flow (Apart from the normal workflow, a system can also have an 'Alternate workflow'. This is the less common interaction done by a user with the system), exception flow (The flow that prevents a user from achieving the goal) and post conditions (The conditions that need to be checked after the case is completed)

In this lecture, the following topics are discussed:

- Use-case analysis-based system testing

Topic- 035:

We explain test case extraction from use cases through an example where we take a very usual example of usage of microwave oven and see if we are to use its embedded software for cooking food, how a test case would be extracted.

In this lecture, the following topics are discussed:

- Use-case analysis-based system testing
- Use-case analysis example

Lesson 15

Specification-based testing

Topic- 036:

Specification Based Testing, refers to the process of testing a program based on what its specification says its behavior should be. In particular, we can develop test cases based on the specification of the program's behavior, without seeing an implementation of the program. Furthermore, we can develop test cases before the program even exists. Specification-based testing allows the tester to make use of program specifications given, and without knowing anything about the ultimate implementation, generate a set of test data that would be sufficient to test a program that would be written in accordance with this specification. There are a variety of ways system specifications can be presented ranging from specifications written in English to mathematical specifications of all functions and underlying data-structures, e.g., system specified using Z, VDM languages or as typed attributed graph transformation system (TAGTS). The methods used for software testing are solely dependent on the application of information which is used to select test data while specifications are accountable for valuable information for testing. The proposed behavior is overlooked completely as the techniques of testing which are based on the implementation emphasize on the real-time performance of the implementation. Based on a requirements specification, this case selection technique consists of test cases selected by the user. As too often in any case, when the specification is informal, the whole thing can be done efficiently.

Test case selection should purely base on source code which has long been recognized. Gourlay has represented a mathematical framework and he acknowledged the need for specification-based testing. A well written and well-defined semantics shall be used to attain results from testing which is based on specification and the semantics must be written in an official language to obtain proper results. Informal specifications will not be able to expose errors; says Laski.

To divide the domain which is used for input data into equivalence classes and to select the data for testing from each of these classes the traditional functional testing method is used. Goodenough and Gerhart enhanced this over-all method. A table known as a condition table is used as it contains several information sources, where columns signifies the test case which is to be derived, which in turn is a blend of conditions which are to be tested.

A technique is provided by Ostrand and Balcer for category-partition in which the person performing the test examines specification and classifies distinct functional units which are testable, inputs of each function are labelled and then it is classified into different categories as per their equivalence. Test case selection approach involves activities related to the documentation interpretation by the testers.

Scholars recommended different practices which concentrates on the specification to select the test cases. It was also suggested that the path domains can be subdivided to construct the subdomains which are likely to be error centered, which may be based on the specifications.

Partition analysis method was devised by Richardson and Clarke which progresses a division by overlaying a partition based on implementation and a specification. Functional testing hires functional decomposition information of design and specification and applies its guidelines to adopt the test cases for several functional classes. From algebraic specifications, an approach is introduced which is used to select the description of categorically enhancing collection of test cases. Constructed on the basis of modification of a predicate calculus specification, a test adequacy technique was proposed by Gopal and Budd. But the mentioned selection of test case tactics have never been defined appropriately which can be applicable in general. The main focus is on testing the specifications rather than their implementation. Goguen and Tardo is in favour of testing the algebraic specifications. Neither of these techniques are focused on the selection of real test cases for the specification.

A tool set used to provide writing capabilities and declarations which are assembled into run-time checks to use in a methodology of debugging is called The Anna tool set. To identify variations in the code Velasco presented a technique that uses assertions supplied by the programmer, which in turn used to choose data for testing.

A special case of test case generation from system specification is when we extract usage scenarios, construct a scenario matrix and execute SUT to do system testing.

In this lecture, the following topics are discussed:

1. Specification-based testing:

- Also known as Behavior Based Testing and Black Box Testing techniques because in this testers view the software as a black-box.
- As they have no knowledge of how the system or component is structured inside the box. In essence, the tester is only concentrating on what the software does, not how it does it.
- Both Functional Testing and Non-Functional Testing is a type of Specification Based Testing.
- Specification Based Test Design Technique uses the specification of the program as the point of reference for test data selection and adequacy. A specification can be anything like a written document, collection of use cases, a set of models or a prototype.

Topic- 037:

Specification Based Testing has a special topic of scenario-based testing where we develop usage scenarios from available specifications. Here, we treat each usage as a meaningful usage story and we develop test cases of system level as well as acceptance level.

In this lecture, the following topics are discussed:

1. Scenario-based testing:

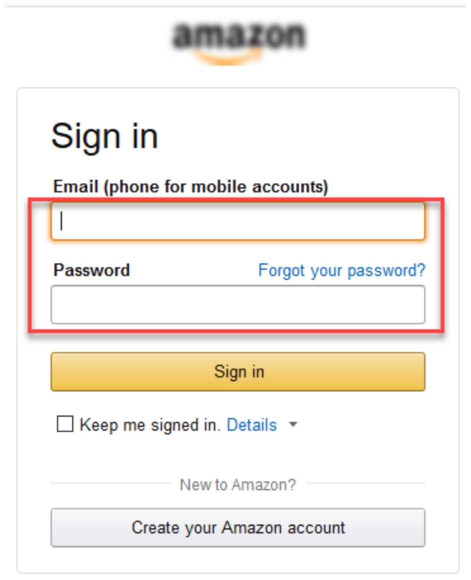
    - Scenario testing is a software testing technique that makes best use of scenarios. Scenarios help a complex system to test better where in the scenarios are to be credible which are easy to evaluate.
    - Methods in Scenario Testing:
        - System scenarios
        - Use-case and role-based scenarios
    - Strategies to Create Good Scenarios:
        - Enumerate possible users their actions and objectives
        - Evaluate users with hacker's mind-set and list possible scenarios of system abuse.
        - List the system events and how does the system handle such requests.
        - List benefits and create end-to-end tasks to check them.
        - Read about similar systems and their behaviour.
        - Studying complaints about competitor's products and their predecessor.

2. Scenario-based testing example

    Test Scenario for eCommerce Application

    For an eCommerce Application, a few test scenarios would be:

    Test Scenario 1: Check the Login Functionality


amazon
Sign in
Email (phone for mobile accounts)
Password
Forgot your password?
Sign in
Keep me signed in. Details
New to Amazon?
Create your Amazon account

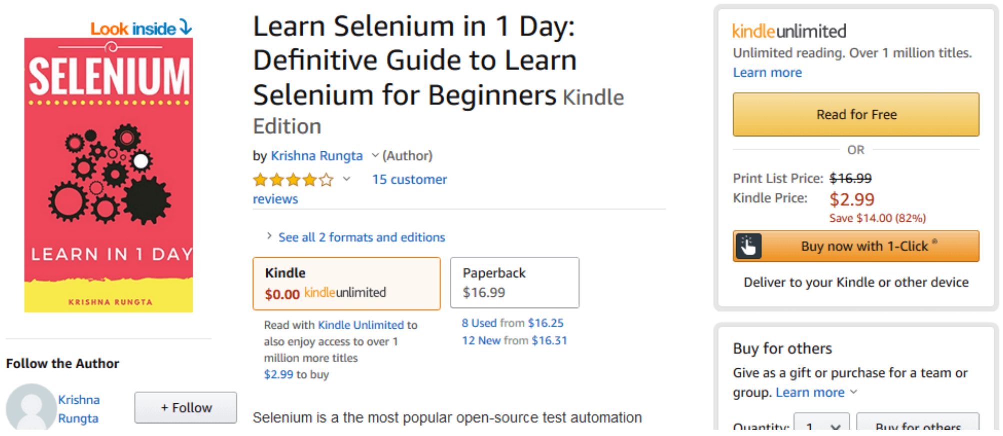
In order to help you understand the difference Test Scenario and Test Cases, specific test cases for this Test Scenario would be

1. Check system behaviour when valid email id and password is entered.
2. Check system behaviour when *invalid* email id and *valid* password is entered.
3. Check system behaviour when *valid* email id and *invalid* password is entered.
4. Check system behaviour when *invalid* email id and *invalid* password is entered.
5. Check system behaviour when email id and password are left blank and Sign in entered.
6. Check Forgot your password is working as expected
7. Check system behaviour when valid/invalid phone number and password is entered.
8. Check system behaviour when "Keep me signed" is checked

As evident, Test Cases are more specific.

Test Scenario 2: Check the Search Functionality

Test Scenario 3: Check the Product Description Page


Look inside

Learn Selenium in 1 Day: Definitive Guide to Learn Selenium for Beginners Kindle Edition

by Krishna Rungta (Author)

15 customer reviews

See all 2 formats and editions

Kindle $0.00 kindleunlimited

Paperback $16.99

Read with Kindle Unlimited to also enjoy access to over 1 million more titles $2.99 to buy

8 Used from $16.25
12 New from $16.31

Follow the Author

Krishna Rungta + Follow

Selenium is a the most popular open-source test automation

kindleunlimited
Unlimited reading. Over 1 million titles. Learn more

Read for Free

OR

Print List Price: ~~$16.99~~
Kindle Price: $2.99
Save $14.00 (82%)

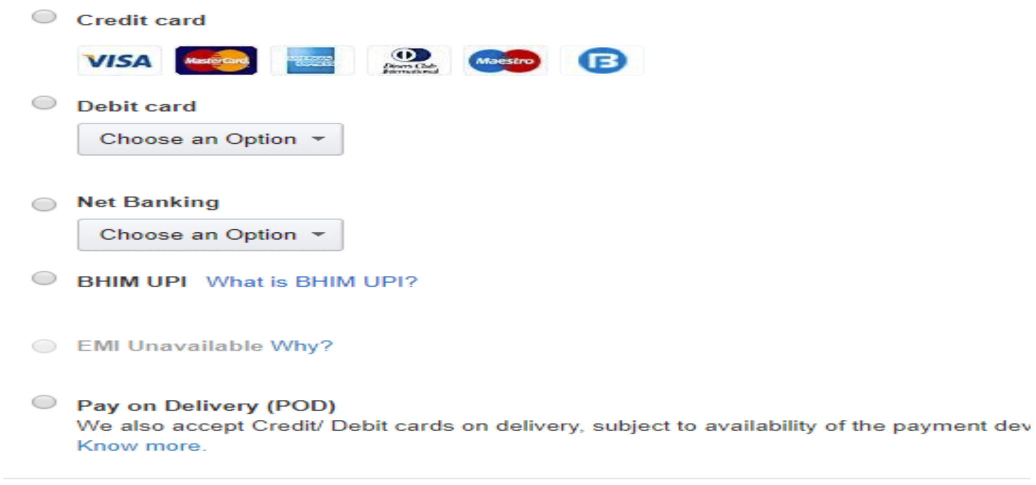
Buy now with 1-Click

Deliver to your Kindle or other device

Buy for others
Give as a gift or purchase for a team or group. Learn more

Quantity: 1 Buy for others

Test Scenario 4: Check the Payments Functionality



Credit card

Debit card

Choose an Option

Net Banking

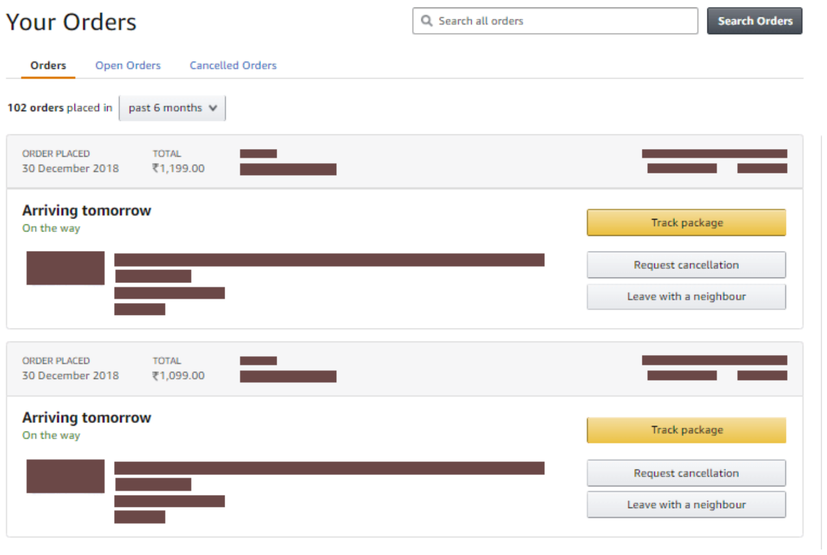
Choose an Option

BHIM UPI What is BHIM UPI?

EMI Unavailable Why?

Pay on Delivery (POD)
We also accept Credit/ Debit cards on delivery, subject to availability of the payment dev
Know more.

Test Scenario 5: Check the Order History



Your Orders

Search all orders | Search Orders

Orders | Open Orders | Cancelled Orders

102 orders placed in past 6 months

ORDER PLACED 30 December 2018 | TOTAL ₹1,199.00 | [redacted]

Arriving tomorrow
On the way

[redacted]

Track package | Request cancellation | Leave with a neighbour

ORDER PLACED 30 December 2018 | TOTAL ₹1,099.00 | [redacted]

Arriving tomorrow
On the way

[redacted]

Track package | Request cancellation | Leave with a neighbour

Apart from these 5 scenarios here is the list of all other scenarios

- Check Home Page behavior for returning customers
- Check Category/Product Pages
- Check Customer Service/Contact Pages
- Check Daily Deals pages

Lesson 16

Equivalence class testing

Topic- 038:

It comes under the Functional Black Box testing technique. As it is a black box testing, there won't be any inspection of the codes. Several interesting facts about this are briefed in this section. It ensures if the path used by the user is working as intended or not. It makes sure that the user can accomplish the task successfully.

Consider a scenario where a user is buying an Item from an Online Shopping Site. The user will First Login to the system and start performing a Search. The user will select one or more items shown in the search results and he will add them to the cart. After all this, he will check out. So this is an Example of logically connected series of steps which the user will perform in a system to accomplish the task. The flow of transactions in the entire system from end to end is tested in this testing. We develop equivalence classes of inputs identified in a way that $I = i_1 \cup i_2 \cup \ldots i_n$ such that $i_1 \cap i_2 \cap \ldots i_n = \Phi$

In this lecture, the following topics are discussed:

1. Equivalence class testing

   Equivalence Partitioning or Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. In this technique, input data units are divided into equivalent partitions that can be used to derive test cases which reduces time required for testing because of small number of test cases.

   - It divides the input data of software into different equivalence data classes.
   - You can apply this technique, where there is a range in the input field.

2. Weak and strong Normal and Robust equivalence

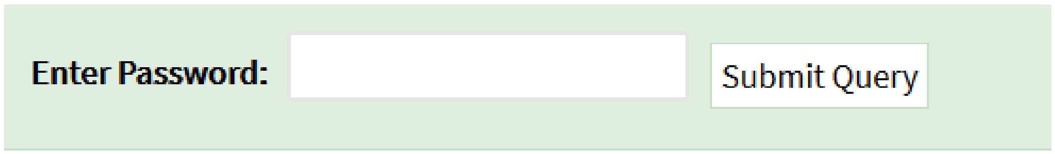
   - Weak Normal Equivalence Class Testing: In this first type of equivalence class testing, one variable from each equivalence class is tested by the team. Moreover, the values are identified in a systematic manner. Weak normal equivalence class testing is also known as **single fault assumption**.
   - Strong Normal Equivalence Class Testing: Termed as multiple fault assumption, in strong normal equivalence class testing the team selects test cases from each element of the Cartesian product of the equivalence. This ensures the notion of completeness

in testing, as it covers all equivalence classes and offers the team one of each possible combinations of inputs.

- Weak Robust Equivalence Class Testing: Like weak normal equivalence, weak robust testing too tests one variable from each equivalence class. However, unlike the former method, it is also focused on testing test cases for invalid values.
- Strong Robust Equivalence Class Testing: Another type of equivalence class testing, strong robust testing produces test cases for all valid and invalid elements of the product of the equivalence class. However, it is incapable of reducing the redundancy in testing.

3. Equivalence class (with boundary value) example

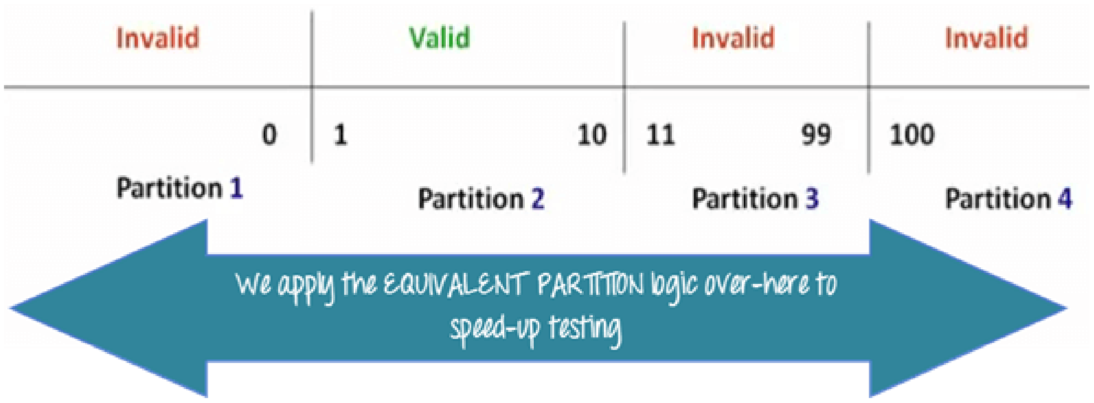
- Let's consider the behavior of Order Pizza Text Box Below
- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, "Only 10 Pizza can be ordered"



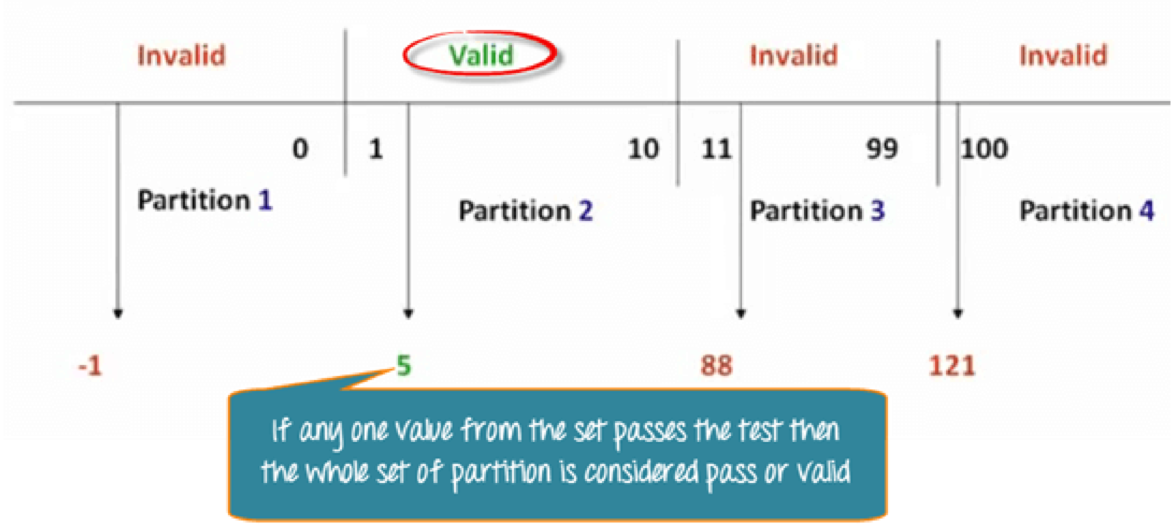
Here is the test condition

1. Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

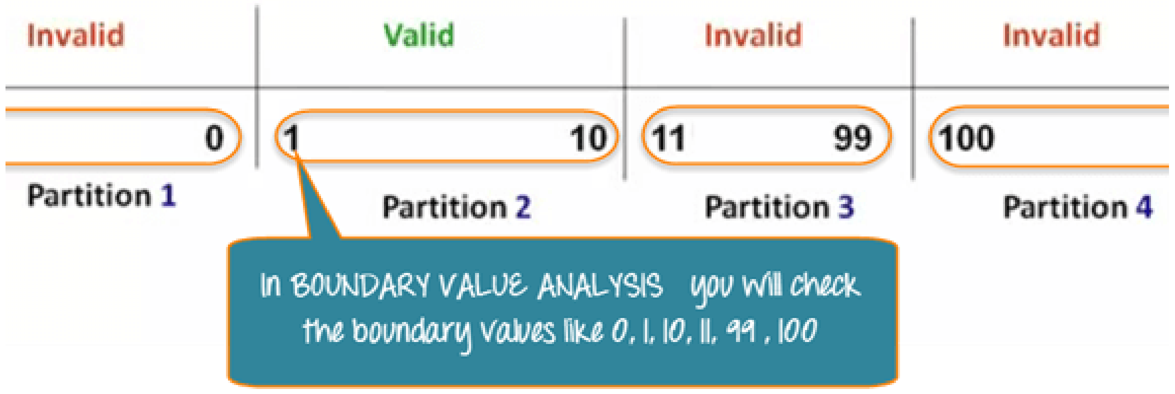
We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behaviour can be considered the same.



The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is that if one condition/value in a partition passes all others will also pass. Likewise, if one condition in a partition fails, all other conditions in that partition will fail.


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
-1
5
88
121
If any one value from the set passes the test then the whole set of partition is considered pass or valid

Boundary Value Analysis- in Boundary Value Analysis, you test boundaries between equivalence partitions


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100

In our earlier example instead of checking, one value for each partition you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at both valid and invalid boundaries. Boundary Value Analysis is also called range checking.

Equivalence partitioning and boundary value analysis (BVA) are closely related and can be used together at all levels of testing.

Lesson 19

Decision table-based testing

Topic- 039:

Decision table testing is a software testing technique used to test system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form. That is why it is also called as a Cause-Effect table where Cause and effects are captured for better test coverage. A Decision Table is a tabular representation of inputs versus rules/cases/test conditions.

1. Decision table-based testing example

Let's learn with an example. The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:

- T – Correct username/password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

Interpretation:

- Case 1 – Username and password both were wrong. The user is shown an error message.
- Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
- Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
- Case 4 – Username and password both were correct, and the user navigated to homepage

While converting this to test case, we can create 2 scenarios ,

- Enter correct username and correct password and click on login, and the expected result will be the user should be navigated to homepage

And one from the below scenario

- Enter wrong username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter correct username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter wrong username and correct password and click on login, and the expected result will be the user should get an error message

As they essentially test the same rule. In this lecture, the following topics are discussed:

- Decision table-based testing
- Decision table testing is a software testing technique used to test system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form. That is why it is also called as a Cause-Effect table where Cause and effects are captured for better test coverage.
- Let's learn with an example.
- Example 1: How to make Decision Base Table for Login Screen
- Let's create a decision table for a login screen.

The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:

- T – Correct username/password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

Interpretation:

- Case 1 – Username and password both were wrong. The user is shown an error message.
- Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
- Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
- Case 4 – Username and password both were correct, and the user navigated to homepage

While converting this to test case, we can create 2 scenarios ,

- Enter correct username and correct password and click on login, and the expected result will be the user should be navigated to homepage

And one from the below scenario

- Enter wrong username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter correct username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter wrong username and correct password and click on login, and the expected result will be the user should get an error message

As they essentially test the same rule.

Lesson 18

Model-based testing

Topic- 040:

Model based testing is a software testing technique where run time behavior of software under test is checked against predictions made by a model. A model is a description of a system's behavior. Behavior can be described in terms of input sequences, actions, conditions, output and flow of data from input to output. It should be practically understandable and can be reusable; shareable must have a precise description of the system under test. There are numerous models available and it describes different aspects of the system behavior. Examples of the model are include data flow, control flow, dependency graph, decision table, state-machine, typed attributed graph transformation systems, etc. Model-Based Testing describes how a system behaves in response to an action (determined by a model). Supply action, and see, if the system responds as per the expectation. It is a lightweight formal method to validate a system. This testing can be applied to both hardware and software testing.

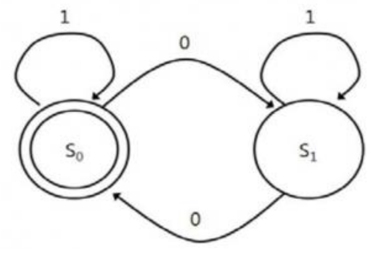
In this lecture, the following topics are discussed:

- Model-based testing

Topic- 041:

State-machine is one such example of system represented as a model. The students here understand how do we represent a system specification and a code in terms of a state-machine learning what is the relevance between specifications / code parts to various aspects of a state-machine.

Finite State Machine is defined formally as a 5-tuple, (Q, Σ, T, q0, F) consisting of a finite set of states Q,a finite set of input symbols Σ, a transition function T: Q x Σ →Q, an initial state q0 ∈ Q, and final states F ⊆ Q .



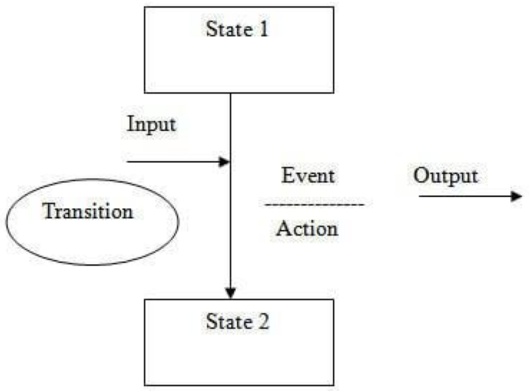
FSM can be described as a state transition diagram. For example, figure 1 depicts state transition diagram where Q= {s0, s1} and Σ = {0, 1}. For complex problems, the difficulty in representing the system as FSM is how to deal with the state explosion problem. A system with n variables that can have Z values can have Zn possible states. FSM is further distinguished by Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA). In DFA, for each pair of state and input symbol there is only one transition to a next state whereas, in NFA, there may be several possible next states. Often NFA refers to NFA-epsilon which allows a transition to a next state without consuming any input symbol. That is, the transition function of NFA is usually defined as T: Q x (ΣU{ε}) → P(Q) where P means power set. Theoretically, DFA and NFA are equivalent as there is an algorithm to transform NFA into DFA. In this lecture, the following topics are discussed:

State-machine for testing

- State Transition testing is a Black-box testing technique, which can be applied to test 'Finite State Machines'.
- A 'Finite State Machine (FSM)' is a system that will be in different discrete states (like "ready", "not ready", "open", "closed",…) depending on the inputs or stimuli.
- The discrete states that the system ends up with, depends on the rules of the transition of the system. That is, if a system gives a different output for the same input, depending on its earlier state, then it is a finite state system.
- Further, if every transaction is tested in the system, it is called "0-switch" coverage. If testing covers 2 pairs of valid transactions, then it is "1-switch" coverage, and so on.

State transition technique is a dynamic testing technique, which is used when the system is defined in terms of a finite number of states and the transitions between the states are governed by the rules of the system.

Or in other words, this technique is used when features of a system are represented as states which transform into one another. The transformations are determined by the rules of the software. The pictorial representation can be shown as:



So here we see that an entity transitions from State 1 to State 2 because of some input condition, which leads to an event and results in action and finally gives the output.

To explain it with an example:

You visit an ATM and withdraw $1000. You get your cash. Now you run out of balance and make exactly the same request of withdrawing $1000. This time ATM refuses to give you the money because of insufficient balance. So, here the transition, which caused the change in state is the earlier withdrawal

State Transition Testing Definition

Having understood what State Transition is, we can now arrive at a more meaningful definition for State Transition testing. So, it is a kind of black-box testing in which the tester has to examine the behavior of AUT (Application Under Test) against various input conditions given in a sequence.

The behavior of the system is recorded for both positive and negative test values.

When to use State Transition Testing?

State Transition testing can be employed in the following situations:

- When the application under test is a real-time system with different states and transitions encompassed.
- When the application is dependent upon the event/values/conditions of the past.
- When the sequence of events needs to be tested.
- When the application needs to be tested against a finite set of input values.

When not to use State Transition Testing?

You should not rely upon State Transition testing under the following situations:

- When testing is not required for sequential input combinations.
- When different functionalities of the application are required to be tested (more like Exploratory testing).

-

Lesson 19

State-machine as a model with example

Topic- 042:

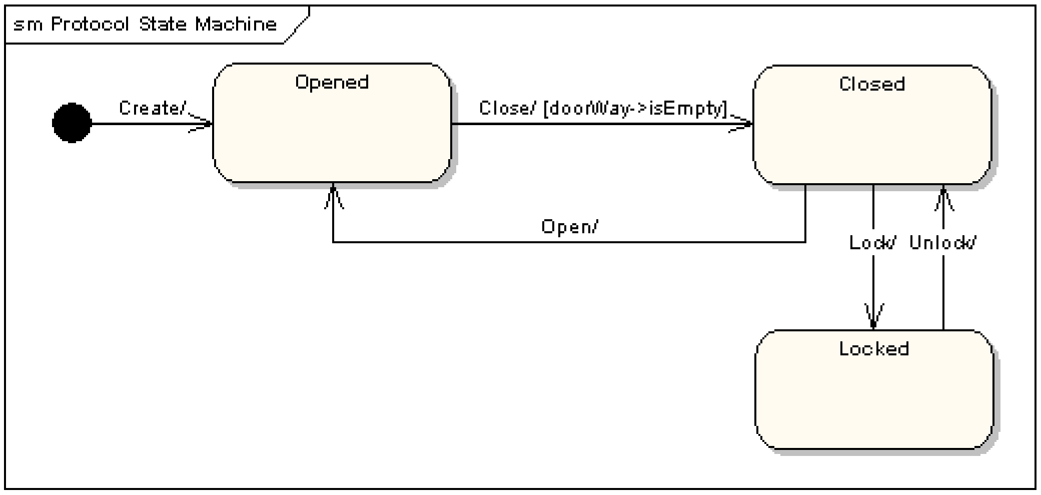
A state machine is a concept used in designing computer programs or digital logic. There are two types of state machines: finite and infinite state machines. The former is comprised of a finite number of states, transitions, and actions that can be modeled with flow graphs, where the path of logic can be detected when conditions are met. The latter is not practically used. State machines are represented using state diagrams. The output of a state machine is a function of the input and the current state. State machines play a significant role in areas such as electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. They are best used in the modeling of application behavior, software engineering, design of hardware digital systems, network protocols, compilers, and the study of computation and languages.

While doing state-machine based testing, we represent the system as a state-machine where the stimulus is provided by inputs and the result is a state transition from A to B. We write test cases that are sets of inputs and expected outputs and our intention is to see if a transition takes place as intended. This gives rise to coverage criteria required specific to this form of model-based testing and a fault model.

In this lecture, the following topics are discussed:

1. State-machine as model – testing

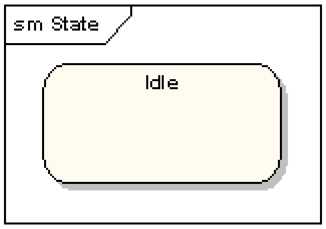
   A state machine diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events. As an example, the following state machine diagram shows the states that a door goes through during its lifetime.



The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled. The syntax and conventions used in state machine diagrams will be discussed in full in the following sections.

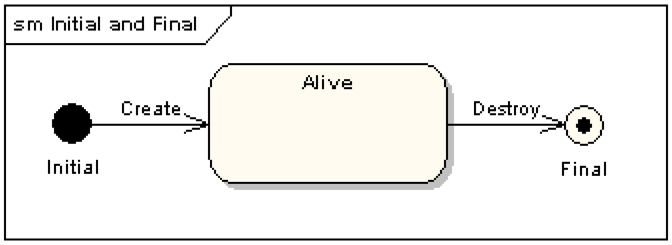
States

A state is denoted by a round-cornered rectangle with the name of the state written inside it.



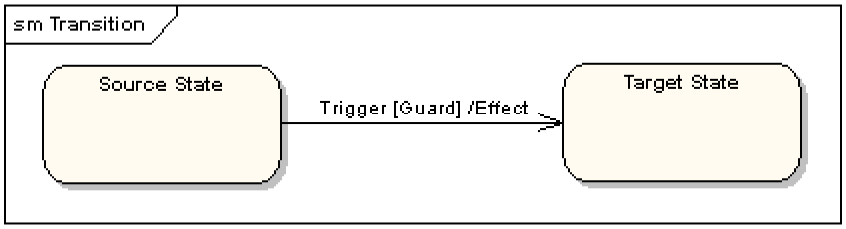
Initial and Final States

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name.



Transitions

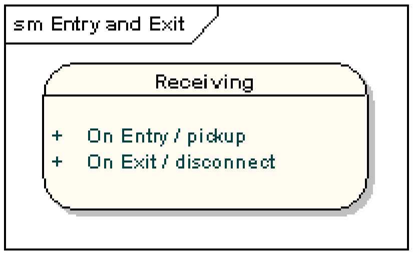
Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions

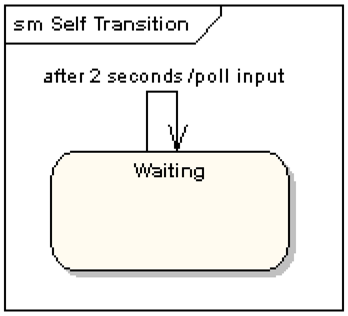
In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

Self-Transitions

A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.



2. State machine-based testing – example

In the practical scenario, testers are normally given the State Transition diagrams and we are required to interpret it. These diagrams are either given by the Business Analysts or a stakeholder and we use these diagrams to determine our test cases. Let’s consider the below situation:

Software name – Manage_display_changes

Specifications – The software responds to input requests to change display mode for a time display device.

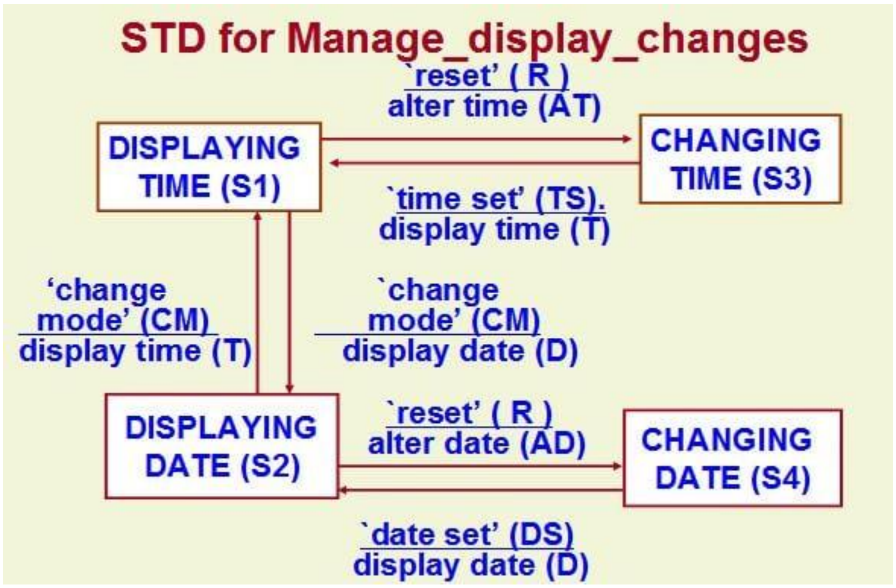
The Display mode can be set to one of the four values:

- Two corresponding to displaying either the time or date.
- The other two when altering either the time or the date.

The different states are as follows:

- Change Mode (CM): Activation of this shall cause the display mode to move between "display time (T)" and "display date (D)".
- Reset (R): If the display mode is set to T or D, then a "reset" shall cause the display mode to be set to "alter time (AT)" or "alter date (AD)" modes.
- Time Set (TS): Activation of this shall cause the display mode to return to T from AT.
- Date Set (DS): Activation of this shall cause the display mode to return to D from AD.

State Transition diagram



Now, let's move to interpret it: Here:

#1) Various States are:

- Display Time(S1),
- Change Time(S3),
- Display Date(S2), and
- Change Date (S4).

#2) Various Inputs are:

- Change Mode(CM),
- Reset (R),
- Time Set(TS),
- Date Set(DS).

#3) Various Outputs are:

- Alter Time(AT),
- Display Time(T),
- Display Date(D),
- Alter Date (AD).

#4) The Changed States are:

- Display Time(S1),
- Change Time (S3),
- Display Date (S2) and
- Change Date (S4).

Step 1: Write all of the start states. For this, take one state at a time and see how many arrows are coming out from it.

- For State S1, there are two arrows coming out of it. One arrow is going to state S3 and another arrow is going to state S2.
- For State S2 – There are 2 arrows. One is going to State S1 and other going to S4
- For State S3 – Only 1 arrow is coming out of it, going to state S1
- For State S4 – Only 1 arrow is coming out of it, going to state S2

Let's put this on our table:

Start State	**S1**	**S1**	**S2**	**S2**	**S3**	**S4**

Since for state S1 and S2, there are two arrows coming out, we have written it twice.

Step -2: For each state, write down their final transitioned states.

- For state S1 – The final states are S2 and S3
- For State S2 – The final states are S1 and S4
- For State S3 – The final state is S1
- For State S4 – Final State is S2

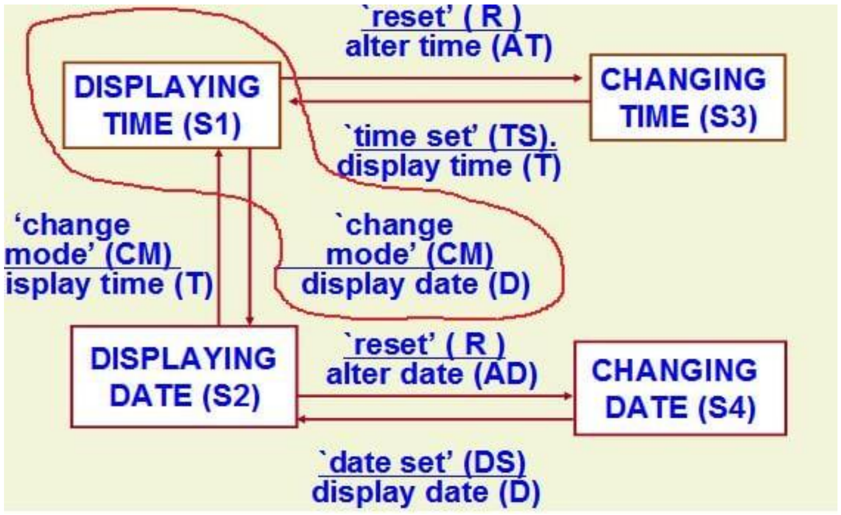
Put this on the table as an Output/Resultant state.

Start State	S1	S1	S2	S2	S3	S4

Finish State	S2	S3	S4	S1	S1	S2

Step 3: For each start state and its corresponding finish state, write down the input and output conditions

For state S1 to go to state S2, the input is Change Mode (CM) and output is Display Date(D) shown below:



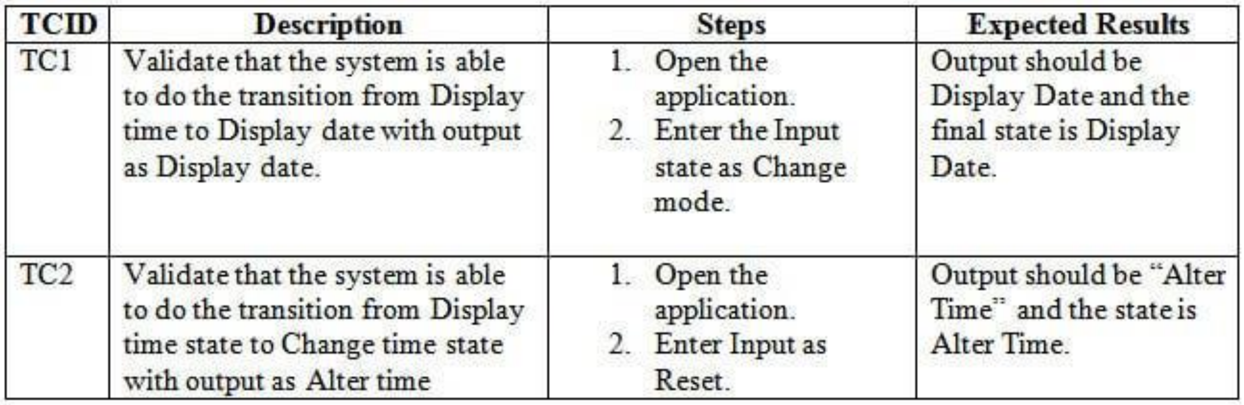
In a similar way, write down the Input conditions and its output for all the states as follows:

Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	R	CM	TS	DS
Output	D	AT	AD	T	T	D
Finish State	S2	S3	S4	S1	S1	S2

Step 4: Now add the test case ID for each test shown below:

Test Case	TC1	TC2	TC3	TC4	TC5	TC6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	R	CM	TS	DS
Output	D	AT	AD	T	T	D
Finish State	S2	S3	S4	S1	S1	S2

Now let's convert it to formal test cases:



TCID	Description	Steps	Expected Results
TC1	Validate that the system is able to do the transition from Display time to Display date with output as Display date.	1. Open the application. 2. Enter the Input state as Change mode.	Output should be Display Date and the final state is Display Date.
TC2	Validate that the system is able to do the transition from Display time state to Change time state with output as Alter time	1. Open the application. 2. Enter Input as Reset.	Output should be "Alter Time" and the state is Alter Time.

In this way, all the remaining test cases can be derived. I assume the other attributes of the test cases like preconditions, severity, priority, environment, build, etc. are also included in the test case.

Lesson 20

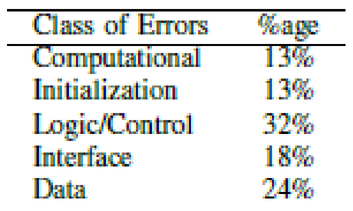
Confidence-based testing

Topic- 043:

In structural testing techniques, we consider the program structure related information to derive test cases. In confidence-based testing methods, we test usefulness of test cases or comprehensiveness of our test suite through a measure of confidence. This means we would have a set of test cases upfront and we would do some form of fault seeding to artificially sow faults. In case, we are doing this one-by-one, we call it mutation testing where we have an original and a mutant version of the program after a fault is introduced into the program. We then run after test suite and see if that introduction makes one of our test cases with "pass" status to fail or report a deviation in the results. In case this happens, we say that our test suite is capable of finding faults and we add a test case to find this sown fault otherwise. If we add a single fault per run, this is called mutation testing and if we do a number of faults in one go, we call it fault seeding.

Now the question arises, what type of faults to be introduced in the system under test. This depends upon fault model that we select. In our lectures, we have considered fault distribution published in and IEEE paper and is:

TABLE I. FAULT DISTRIBUTION



Class of Errors	%age
Computational	13%
Initialization	13%
Logic/Control	32%
Interface	18%
Data	24%

In this lecture, the following topics are discussed:

1. Confidence-based testing
2. Mutation testing

Mutation testing is a fault-based testing technique where variations of a software program are subjected to the test dataset. This is done to determine the effectiveness of the test set in isolating the deviations**.** Say, there is a hospital site that lets new users register. It reads the Date of birth or age of the patient. If it is greater than 14, assigns a general physician as their main doctor. To do so, it invokes the 'general physician' function that finds the available doctor.

Now, there might be other functionality. Maybe, patients below 13 get assigned to a pediatrician and so on. But we will only take the age-over-14 case. This is what the code might look like:

```
1) Read Age
2) If age>14
3) Doctor= General Physician()
4) End if
```

Please note that the above lines of code are not specific to any programming language and won't run. It is just hypothetical. As a tester, if my data-set is 14, 15, 0, 13 – some random numbers. The target is to check if the data-set of the 4 values (14, 15, 0, and 3) is adequate to identify all possible problems with this code.

How does Mutation Testing achieve this?

First and foremost, you create mutants- variations of the program. A mutant is nothing but a program that is written as a deviation. It contains a self-seeded fault.

Examples are:

- Arithmetic operator replacement
- Logical connector replacement
- Statement removal
- Relational operator replacement
- Absolute value insertion, etc.

These replacements are also called 'Mutation Operators.'

Let us consider examples:

Mutant #1: Relational operator replacement

```
1) Read Age
2) If age<14 'Changing the > with <'
3) Doctor= General Physician()
4) End if
```

Mutant #2:

```
1) Read Age
2) If age=14 'Changing the > with ='
3) Doctor= General Physician()
4) End if
```

Mutant #3:

```
1) Read Age
2) If age>=14 'Changing the > with >='
3) Doctor= General Physician()
4) End if
```

Mutant #4:

```
1) Read Age
2) If age<=14 'Changing the > with <='
3) Doctor= General Physician()
4) End if
```

Mutant #5: Statement Removal

```
1) Read Age
2) If age=14
3) 'remove the doctor assignment statement'
4) End if
```

Mutant #6: Absolute Value Insertion

```
1) Read Age
2) If age>14
3) Doctor= Mr.X (Absolute value insertion- let's say X is a pediatrician)
4) End if
```

Mutant #7: Incorrect syntax

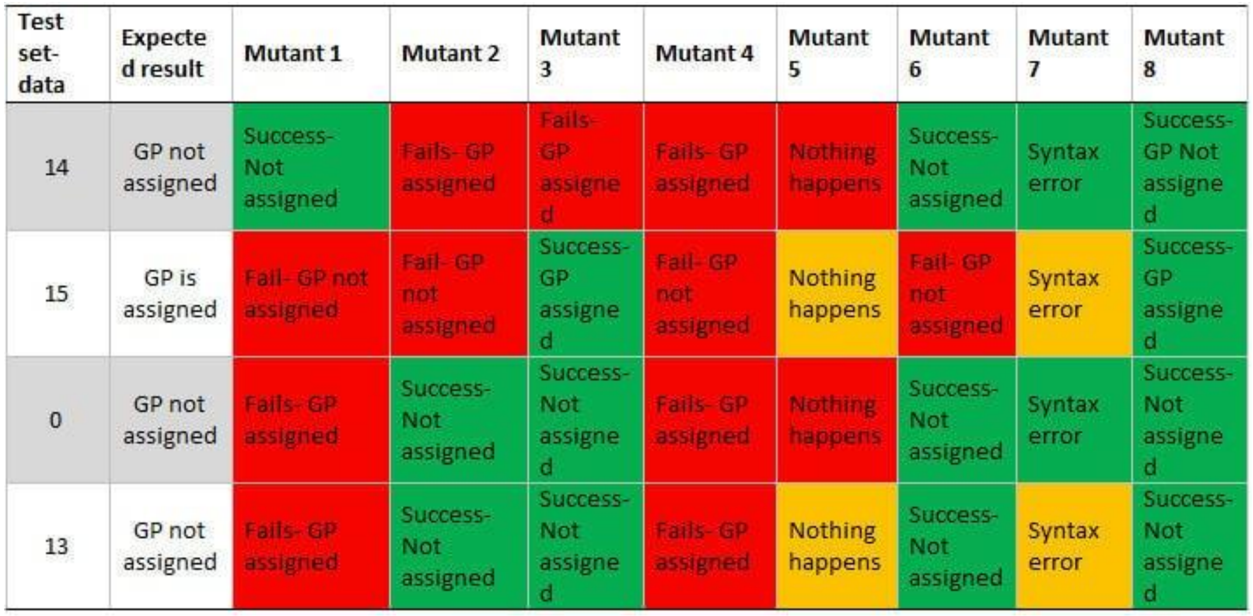
```
1) Read Age
2) If age%%14 (incorrect syntax)
3) Doctor=General Physician()
4) End if
```

Mutant #8: Does the same thing as the original test

```
1) Read Age
2) If age> 14 & age>14 'means the same thing as age>14'
3) Doctor= General Physician()
4) End if
```

Once, all the mutants are created. They are subjected to the test data-set. Our set is 14, 15, 0 and 13. Which of these mutants will our data-set find?

Find out in the below table:
(Click on image for an enlarged view)



Test set-data	Expected result	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 5	Mutant 6	Mutant 7	Mutant 8
14	GP not assigned	Success- Not assigned	Fails- GP assigned	Fails- GP assigned	Fails- GP assigned	Nothing happens	Success- Not assigned	Syntax error	Success- GP Not assigned
15	GP is assigned	Fail- GP not assigned	Fail- GP not assigned	Success- GP assigned	Fail- GP not assigned	Nothing happens	Fail- GP not assigned	Syntax error	Success- GP assigned
0	GP not assigned	Fails- GP assigned	Success- Not assigned	Success- Not assigned	Fails- GP assigned	Nothing happens	Success- Not assigned	Syntax error	Success- Not assigned
13	GP not assigned	Fails- GP assigned	Success- Not assigned	Success- Not assigned	Fails- GP assigned	Nothing happens	Success- Not assigned	Syntax error	Success- Not assigned

As you can see our data value 14 finds failures when it runs against, Mutant 2, 3 and 4. Or, 14 kills mutants 2, 3 & 4. But, it is ineffective against, 1, 6 and 8.

If your data-set kills all mutants, it is effective. Otherwise, include more or better test data. It is not necessary for the each value in the data-set to kill all mutants. But together, they should kill all. For example: 14 kills 2, 3 and 4. 15 kills 1, 2 and 4. And, so on.

What about 5, 7, and 8?

Mutant #5 – is the program instance that will fail irrespective of any data value you give. This is because it will not do any programming for both valid and invalid values.

Mutant #7 – will be a compile error. Or in the case of a scripting language an error that will prevent execution.

Mutant #8 – is the same thing as the main program.

As you can see, the above mutants are not useful at all.

Therefore, mutants to avoid are:

- Syntactically incorrect/'Still-Born' mutants. : You need syntactically correct mutants ONLY. Example: Mutant 7
- Equivalent Mutants: The ones that do the exact same thing as the original program. Example: Mutant 8.
- Trivial Mutant: Can be killed by any data-set. Example: Mutant 5

Points to note:

- The number of mutants, even for a small program, can be many. It is a finite number, but still very large. Due to this, a subset of mutants is usually used. It is common to choose the mutants randomly.
- The mutation operators listed above is not an exhaustive list. There can be many other variations. I have oversimplified the concept for easy understanding.
- The mutation operators also differ with programming languages, design, and specifications of the application.
- If there are some mutants alive at the end of the test. It means either it is an invalid mutant (like 5, 7 and 8) or the data-set was inadequate. If it is the later one, go back and change it.
- Mutation Test is a structural, white-box and unit testing method. It uses fault-injection or fault-seeding to generate its mutants.
- There are many unit testing frameworks and tools that aid for automatic mutation testing. Some of them are:
    - Jester for JUnit.
    - Pester for Python
    - MuClipse for eclipse, etc.

Are you thinking, if it takes this much effort, what is going to happen when I have to test large samples of code?

Mutation testing relies on two things:

- Competent Programmer Assumption: If a programmer is competent enough to write and test a small piece of code, he will be good at writing larger programs too.
- Coupling effect Assumption: If 2 units are combined to form a program and each one is good in itself, then the combination is going to be good too.

So, it focuses on the smallest unit of code and places its faith in the programmer's skill to scale mutation testing to larger programs.

Topic- 044:

The effectiveness of fault-based testing depends on the quality of the fault model, and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program. Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure programmer hypothesis detection of all such faults. This is known as the competent programmer hypothesis, an assumption that the program under test is "close to" (in the sense of textual difference) a correct program

1. Test and mutation adequacy

Lesson 21

Fault Seeding

Topic- 045:

We take the following steps for implementing fault seeding approach. We "seed" the program with a number of "typical errors" keeping careful track of the changes made before testing and we run test cases to uncover those faults that have been seeded. In the process, we can also hit faults that we not previously uncovered along with those that have been artificially seeded. Therefore, after a period of testing, we compare the number of seeded and non-seeded errors detected. The errors are seeded following a fault model or a fault classification and after a period of testing, compare the number of seeded and non-seeded errors detected.

The effectiveness of fault-based testing depends on the quality of the fault model, and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program. Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure programmer hypothesis detection of all such faults. This is known as the competent programmer hypothesis, an assumption that the program under test is "close to" (in the sense of textual difference) a correct program

In this lecture, the following topics are discussed:

1. Fault seeding

Fault Seeding

Fault seeding is a technique for evaluating the effectiveness of a testing process.

- One or more faults are deliberately introduced into a code base, without informing the testers.
- The discovery of seeded faults during testing can be used to calibrate the effectiveness of the test process.
- Let S is the total number of seeded faults, and $s(t)$ is the number of seeded faults that have been discovered at time t.
- $s(t)/S$ is the seed-discovery effectiveness of testing to time t.
- If seeded faults are assumed are to be representative of actual faults, then seed-discovery effectiveness can be assumed to be representative of overall testing effectiveness.

2. Mutation testing example

Mutation Adequacy

Mutation adequacy uses a similar concept to fault seeding to evaluate the effectiveness of a test suite.

- Assume we have a test suite *TS* with *C* total test cases *c*(*j*).
- Assume that the program under test *P* passes all the test cases *c*(*j*) for 1 <= *j* <= *C*.
- Can we stop testing? That is, have we tested *P* adequately?
- The mutation adequacy criterion provides one answer that we might use.

The mutation adequacy approach differs from fault seeding in that it is applied at a particular point in the testing process and also in that faults are not directly inserted into *P*.

- Instead, a series of *mutants m*(*i*) are created.
- Each mutant *m*(*i*) differs from *P* by the injection of exactly one fault.
- Let *M* be the total number of mutants *m*(*i*).
- The test suite *TS* is applied to each mutant *m*(*i*).
- If a particular mutant *m*(*i*) fails any test in *c*(*j*), then it is said to be *killed.*
- All mutants that are not killed are said to remain *live* at this point.
- The ratio of killed to total mutants (*K*/*M*) can be considered a measure of adequacy of *TS.*

Automated Mutation: Mutation Operators

- Manually creating mutants is time-consuming.
- A collection of mutants *m*(*i*) created from *P* at some point in time will no longer be representative of *P* after it has undergone many changes.
- Mutation can be automated by through the concept of *mutation operators*.
- Mutation operators are simple changes that can be made at various program locations.

Some Mutation Operators

Mutation Operator	Meaning	Original Code	Mutated Code
Add 1	Add 1 to a constant	q = 0;	q = 1;
Replace Variable	Replace a variable with a different one of the same type	r = x;	r = y;
Replace Operator	Replace an operator with a compatible one	q = q + 1	q = q - 1

A Program and Three Mutants

The following table shows a program *P* to perform integer division. Given inputs x and y, *P* computes the integer division of x divided by y producing quotient q and remainder r. Three mutants *m*(1), *m*(2), and *m*(3) are shown resulting from application of the mutation operators in the previous table.

P	*m*(1)	*m*(2)	*m*(3)

q = 0;
 	
r = x;
 	
while r >= y {
 	
r = r - y;
 	
q = q + 1;
 	
}
	q = 1;
	
	r = x;
	
	while r >= y {
	
	r = r - y;
	
	q = q + 1;
	
	}
		q = 0;
		
		r = y;
		
		while r >= y {
		
		r = r - y;
		
		q = q + 1;
		
		}
			q = 0;
			
			r = x;
			
			while r >= y {
			
			r = r - y;
			
			q = q - 1;
			
			}

Mutation Theory

Read the Mutation Theory section of the Mutation Testing Repository for insight into the following topics.

Competent Programmer Assumption

Coupling Effect

Problems of Mutation Adequacy

Equivalent Mutants

Computational Cost

3. Fault seeding example

Sample Problem

- Seed 100 faults into a project at time 0.
- Testing continues to time 30, at which point 73 of the seeded faults have been detected.
- If 219 actual faults were discovered, what is the expected number of total faults prior to seeding?
- How many latent faults are expected to remain in the software at time 30?

- Answer:
  - The discovered original faults are three times the numbered of discovered seeded faults, so 300 original faults are expected.
  - The latent faults are those remaining and not removed: (300 - 219) original faults plus (100 - 73) seeded faults, that is 81 + 27 = 108 latent faults.

Lesson 22

Introduction to software verification

Topic- 045:

Verification is the process of evaluating work-products different phases of software development lifecycle (SDLC) to determine whether they meet the specified requirements. Verification is processing of ensuring that the product is built according to the requirements and design specifications. It also answers to famous question of "Are we building the product right?" as introduced by Ian Sommerville.

There are various methods for software verification that we discuss in this lecture that include peer reviews, walkthroughs and inspections In this lecture, the following topics are discussed:

1. Introduction to software verification

   Verification in Software Testing is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of verification process is to ensure quality of software application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.

   Validation in Software Testing is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. The validation process involves activities like unit testing, integration testing, system testing and user acceptance testing.

   Difference:

   - Verification process includes checking of documents, design, code and program whereas Validation process includes testing and validation of the actual product.
   - Verification does not involve code execution while Validation involves code execution.
   - Verification uses methods like reviews, walkthroughs, inspections and desk-checking whereas Validation uses methods like black box testing, white box testing and non-functional testing.
   - Verification checks whether the software confirms a specification whereas Validation checks whether the software meets the requirements and expectations.
   - Verification finds the bugs early in the development cycle whereas Validation finds the bugs that verification can not catch.
   - Verification process targets on software architecture, design, database, etc. while Validation process targets the actual software product.
   - Verification is done by the QA team while Validation is done by the involvement of testing team with QA team.

- Verification process comes before validation whereas Validation process comes after verification.

2. Verification methods

In the context of testing, “Verification and Validation” are the two widely and commonly used terms. Most of the times, we consider both the terms as the same, but actually, these terms are quite different.

There are two aspects of V&V (Verification & Validation) tasks:

- Confirms to requirements (Producer view of quality)
- Fit for use (consumers view of quality)

Producer’s view of quality, in simpler terms, means the developers perception of the final product.
Consumers view quality means the user’s perception of the final product. When we carry out the V&V tasks, we must concentrate on both of these views of quality. Let us first start with the definitions of verification and validation and then we will go about understanding these terms with examples. Verification is the process of evaluating the intermediary work products of a software development lifecycle to check if we are in the right track of creating the final product. In other words, we can also state that verification is a process to evaluate the mediator products of software to check whether the products satisfy the conditions imposed during the beginning of the phase. Now the question here is: What are the intermediary or mediator products? Well, these can include the documents which are produced during the development phases like, requirements specification, design documents, database table design, ER diagrams, test cases, traceability matrix, etc. We sometimes tend to neglect the importance of reviewing these documents, but we should understand that reviewing itself can find out many hidden anomalies when if found or fixed in the later phase of the development cycle, can be very costly. *Verification ensures that the system (software, hardware, documentation, and personnel) complies with an organization’s standards and processes, relying on the review or non-executable methods.*

Where is Verification Performed?

Specific to IT projects, following are some of the areas (I must emphasize that this is not all) in which verification is performed.

Verification Situation	**Actors**	**Definition**	**Output**
Business/Functional Requirement Review	Dev team/client for business requirements.	This is a necessary step to not only make sure that the requirements have been	Finalized requirements that are ready to be

Verification Situation	**Actors**	**Definition**	**Output**
		gathered and/or correctly but also to make sure if they are feasible or not.	consumed by the next step – design.
Design Review	Dev team	Following the design creation, the Dev team reviews it thoroughly to make sure that the functional requirements can be met via the design proposed.	Design is ready to be implemented into an IT system.
Code Walkthrough	Individual Developer	The code once written is reviewed to identify any syntactic errors. This is more casual in nature and is performed by the individual developer on the code developed by oneself.	Code ready for unit testing.
Code Inspection	Dev team	This is a more formal set up. Subject matter experts and developers check the code to make sure it is in accordance with the business and functional goals targeted by the software.	Code ready for testing.
Test Plan Review (internal to QA team)	QA team	A test plan is internally reviewed by the QA team to make sure that it is accurate and complete.	A test plan document ready to be shared with the external teams (Project Management, Business Analysis, development, Environment, client, etc.)
Test Plan Review (External)	Project Manager, Business Analyst, and Developer.	A formal analysis of the test plan document to make sure that the timeline and other considerations of the QA team are in line with the other teams and the entire project itself.	A signed off or approved test plan document based on which the testing activity is going to be based on.

Verification Situation	Actors	Definition	Output
Test documentation review (Peer review)	QA team members	A peer review is where the team members review one another's work to make sure that there are no mistakes in the documentation itself.	Test documentation ready to be shared with the external teams.
Test documentation final review	Business Analyst and development team.	A test documentation review to make sure that the test cases cover all the business conditions and functional elements of the system.	Test documentation ready to be executed.

See the test documentation review article which posts a detailed process on how testers can perform the review.

What Is Validation?

Validation is the process of evaluating the final product to check whether the software meets the business needs. In simple words, the test execution which we do in our day to day life is actually the validation activity which includes smoke testing, functional testing, regression testing, systems testing, etc. Validation is all forms of testing that involves working with the product and putting it to test.

Given below are the validation techniques:

- Unit Testing
- Integration testing
- System Testing
- User Acceptance Testing

Validation physically ensures that the system operates according to a plan by executing the system functions through a series of tests that can be observed and evaluated.

- Validating the production issues.

Difference Between Verification And Validation

Verification	Validation
Evaluates the intermediary products to check whether it meets the specific requirements of the particular phase.	Evaluates the final product to check whether it meets the business needs.

Verification	**Validation**
Checks whether the product is built as per the specified requirement and design specification.	It determines whether the software is fit for use and satisfies the business needs.
Checks “Are we building the product right”?	Checks “Are we building the right product”?
This is done without executing the software.	Is done with executing the software.
Involves all the static testing techniques.	Includes all the dynamic testing techniques.
Examples include reviews, inspection, and walkthrough.	Example includes all types of testing like smoke, regression, functional, systems and UAT.

Lesson 23

Walkthroughs and Peer Reviews

Topic- 047:

Walkthrough are used for software verification to review documents with peers, managers, and fellow team members who are guided by the author of the document to gather feedback and reach a consensus. A walkthrough can be pre-planned or organized based on the needs. This review becomes more beneficial for those people who are away from software sphere and through this meeting they get a good insight about the product that is to be developed.

We select members of the teams from different backgrounds in order to have a diverse point of view and thus, provide different dimensions to a common objective. It is pertinent to mention that this this is not a formal process and it is used to find out if there are any misunderstandings or deviations from established standards both informal or defined that can be corrected or avoided. This is helpful in avoiding any manifestation of faults from mistakes made in the earlier stages such as requirements specifications, design and development and even testing process itself.

In this lecture, the following topics are discussed:

1. Walkthrough

Topic- 048:

Peer review or testing is a way of evaluating work performed by a co-worker. In software development, it stands for cross-checking the code written by developers. In review process, various things come to surface such as future casualties, technical content, quality, specifications of the software, etc. This deep overview of a software gives an idea that how will the software run and is shortcomings that are likely to occur in the near future. It also provides an idea that this software is even worth launching. Sometimes, the software has fewer benefits and greater number of disadvantages falling, in this condition the software is discarded or a better development of better software is proposed.

Peer review can be characterized as a method which can be review of documents (called artifacts) code, pair-programming, or a technical review by an expert who takes help from formal checklists to conduct this review as a process.

Peer review requires authors to present their work, involves a team of 2 to 7 professionals where the report optional to author and it presents issues in the shape a few faults. At the end of this process, there are questions such as what are types of issues and how their redressal can be planned, how these issues or faults can impact desired results and how these undesired outcomes can be avoided.

In this lecture, the following topics are discussed:

1. Peer review

Lesson 24

Inspections

Topic- 049:

Inspection is a formal process which involves a team with mandate and preparation to do inspections. Such teams have a plan to undertake and have a list of predefined steps. There are various step-patterns are identified and published by a number of researchers.

There are predefined steps involved in carrying out inspections and since this is a formal process, it requires a kind of formal documentation which we call checklists.

In this lecture, the following topics are discussed:

1. Inspection for verification

Topic- 050:

One such pattern is proposed by Fagan inspection methods where different roles and responsibilities are identified. Author is the person that creates a product that should be inspected whereas moderator is the main person in the process. Team carrying out inspection creates the inspection plan and plays the role of the coordinator of the whole working group.

Besides, team performs the duty of chairman in a session where issues about recording of errors are discussed. Fagan inspection also identifies a reader whose role is only to read to the other members of inspection commission the text of documents that will be inspected. In order to formally record the process, we identify recorder as a role who documents all the errors that were found by the working group. A software tester can play a role of recorder. He/she also compiles a list of issues that are not discussed during the testing session.

These issues can be not only about the found errors. Sometimes the recorder should transfer information from the members of the working group to the other project members and to the members of working group that for some reason were not present at the inspection session.

There are following steps of the inspection process:

1. Planning
2. Overview (1-to-n meeting)

3. Preparation (individual inspection)
4. Inspection (n-to-n meeting)
5. Rework
6. Follow-up

In this lecture, the following topics are discussed:

1. Inspection for verification
2. Fagan inspection for verification

Lesson 25

Example of Inspections / Reviews using Checklists

Topic- 051:

As defined earlier;

- Walkthrough are used for software verification to review documents with peers, managers, and fellow team members who are guided by the author of the document to gather feedback and reach a consensus. A walkthrough can be pre-planned or organized based on the needs. This review becomes more beneficial for those people who are away from software sphere and through this meeting they get a good insight about the product that is to be developed.
- Peer review can be characterized as a method which can be review of documents (called artifacts) code, pair-programming, or a technical review by an expert who takes help from formal checklists to conduct this review as a process.
- Inspection is a formal process which involves a team with mandate and preparation to do inspections. Such teams have a plan to undertake and have a list of predefined steps. There are various step-patterns are identified and published by a number of researchers. One such pattern is proposed by Fagan inspection methods where different roles and responsibilities are identified. Author is the person that creates a product that should be inspected whereas moderator is the main person in the process. He/she creates the inspection plan and plays the role of the coordinator of the whole working group.

We need to know how the theoretical standpoint gets transformed into practically undertaking these steps as a means to conduction verification. For that, we introduce students with examples of checklists for SRS, code and documentation. This is however to explain that checklists are not limited to SRS, code and documentation only, they cover a broad range of software artifacts that are associated with SDLC.

In this lecture, the following topics are discussed:

1. Examples of checklists for SRS document inspections
2. Example of checklist for code inspections
3. Checklist for user documentation inspection
4. Code review examples

Lesson 26

Test Planning

Topic- 052:

Software testing has an important aspect of planning and executing the whole activity following a proper plan. test plan: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice and any risks requiring contingency planning.

It is a record of the test planning process. There are several documents and sub-documents that are associated with test planning. Master test plan not only show the test plan but it also has test management plan as well. We define sub plan that is associated with part of the system or module as phase test plan which could be associated with phases such as unit test plan, integration test plan or acceptance test plan.

In this lecture, the following topics are discussed:

1. Test Planning

   Test planning, the most important activity to ensure that there is initially a list of tasks and milestones in a baseline plan to track the progress of the project. It also defines the size of the test effort.

   It is the main document often called as master test plan or a project test plan and usually developed during the early phase of the project.

Test Plan Contents - Sample:

S.No.	Parameter	Description
1.	Test plan identifier	Unique identifying reference.

2.	Introduction	A brief introduction about the project and to the document.
3.	Test items	A test item is a software item that is the application under test.
4.	Features to be tested	A feature that needs to tested on the testware.
5.	Features not to be tested	Identify the features and the reasons for not including as part of testing.
6.	Approach	Details about the overall approach to testing.
7.	Item pass/fail criteria	Documented whether a software item has passed or failed its test.
8.	Test deliverables	The deliverables that are delivered as part of the testing process,such as test plans, test specifications and test summary reports.
9.	Testing tasks	All tasks for planning and executing the testing.
10.	Environmental needs	Defining the environmental requirements such as hardware, software, OS, network configurations, tools required.
11.	Responsibilities	Lists the roles and responsibilities of the team members.
12.	Staffing and training needs	Captures the actual staffing requirements and any specific skills and training requirements.
13.	Schedule	States the important project delivery dates and key milestones.
14.	Risks and Mitigation	High-level project risks and assumptions and a mitigating plan for each identified risk.
15.	Approvals	Captures all approvers of the document, their titles and the sign off date.

Topic- 053:

A test plan has identified, introduction, features or items to be tested, items not to be tested, reference to other plans such as project plan, configuration management plan, test items, approach, item pass / fail criteria, suspension, resumption and testing deliverables. It identifies how the bugs log would be reported, how test environment would be setup and what would be the schedule of overall activity. The plan is a concise document avoiding redundancy and superfluous-ness.

In this lecture, the following topics are discussed:

1. Test plan document is used to determine:

- Scope and the risks that need to be tested and that are NOT to be tested.
- Documenting Test Strategy.
- Making sure that the testing activities have been included.
- Deciding Entry and Exit criteria.
- Evaluating the test estimate.
- Planning when and how to test and deciding how the test results will be evaluated, and defining test exit criterion.
- The Test artefacts delivered as part of test execution.
- Defining the management information, including the metrics required and defect resolution and risk issues.

2. Test plan objectives and motivation

Topic- 054:

Test case has a certain lifecycle where a test case is identified, it is loaded with runtime data, it is ready for execution and it is finally executed. Once executed, it goes either to pass or to fail state. If it fails, it can block some of the test cases rendering the testing in halt state. This is explained with the help of diagrams and state-charts.

1. Test case lifecycle

Lesson 27

Test suite structure and test reporting systems

Topic- 055:

We have a test suite structure which is not mandatory rather it is preferred to have a hierarchical structure where we have test cases separated according to concerns. Similar to test case lifecycle, we have software testing lifecycle phases. Once the testing activity is complete, we do test result reporting in a manner that we have a report to deliver against each anomaly recorded during testing and an overall stat or the result showing net results. The topics discussed in this lecture are pertaining to test suite structure, software testing lifecycle phases, test results reporting and bug log details.

In this lecture, the following topics are discussed:

1. Test suite structure
2. Software testing lifecycle phases

Topic- 056:

Testing process is completed with test reporting cycle following test execution is complete. We need to understand how a fault is reported, how it is replicated, how bug log form is filled and how a detailed report is prepared.

In this lecture, the following topics are discussed:

1. Test result reporting
2. Bug tracking databases

Lesson 28

Test Management

Topic- 057:

Test management, process of managing the tests. A test management is also performed using tools to manage both types of tests, automated and manual, that have been previously specified by a test procedure. Test management tools allow automatic generation of the requirement test matrix (RTM), which is an indication of functional coverage of the application under test (SUT). Test Management tool often has multifunctional capabilities such as testware management, test scheduling, the logging of results, test tracking, incident management and test reporting.

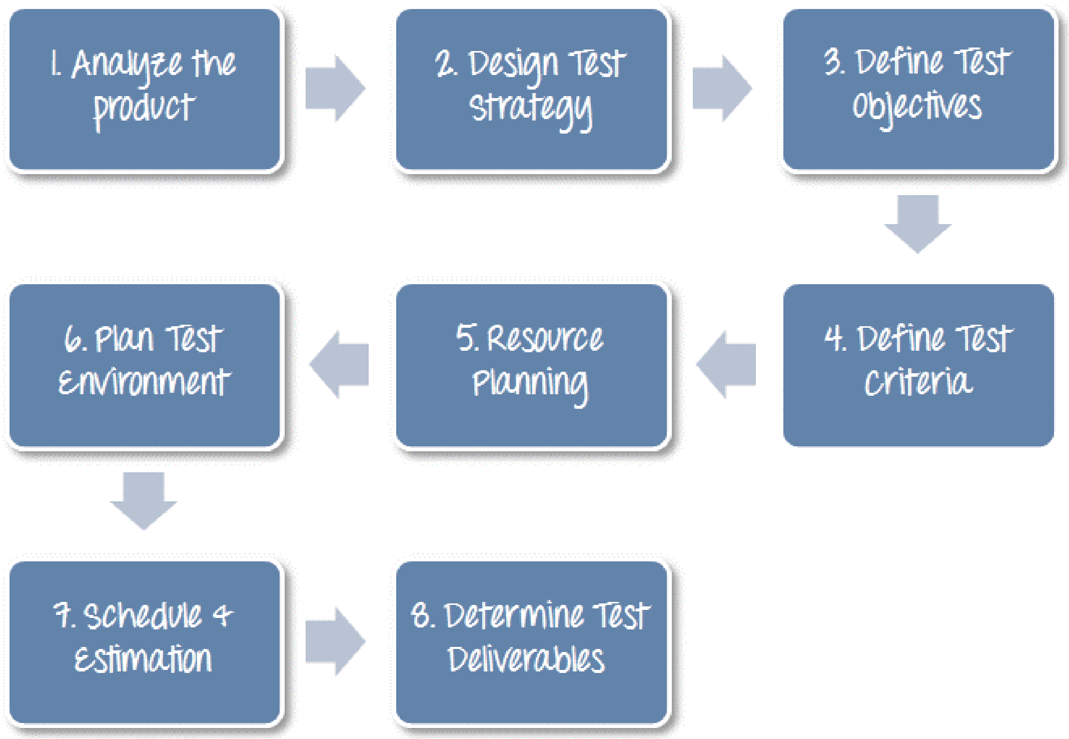
Test Management has a clear set of roles and responsibilities for improving the quality of the product. Test management helps the development and maintenance of product metrics during the course of project. Test management enables developers to make sure that there are fewer design or coding faults. Test management has the following set of activities or the processes involved:

1. Test Plan development

- Ensuring that the test documentation generates repeatable test assets.

3. How to write a Test Plan

- You already know that making a Test Plan is the most important task of Test Management Process. Follow the seven steps below to create a test plan as per IEEE 829
- Analyze the product
- Design the Test Strategy
- Define the Test Objectives
- Define Test Criteria
- Resource Planning
- Plan Test Environment
- Schedule & Estimation
- Determine Test Deliverables



2. Test design

- By design we mean to create a plan for how to implement an idea and technique is a method or way for performing a task. So, Test Design is creating a set of inputs for given software that will provide a set of expected outputs. The idea is to ensure that the system is working good enough and it can be released with as few problems as possible for the average user.
- Test design is a significant step in the Software Development Life Cycle (SDLC), also known as creating test suites or testing a program. In other words, its primary purpose is to create a set of inputs that can provide a set of expected outputs, to address these concerns:
- What to test and what not to test
- How to stimulate the system and with what data values
- How the system should react and respond to the stimuli
- Broadly speaking there are two main categories of Test Design Techniques. They are:
- Static Techniques
- Dynamic Techniques

3. **Text execution**

- Test Objective is the overall goal and achievement of the test execution. The objective of the testing is finding as many software defects as possible; ensure that the software under test is bug free before release.
- To define the test objectives, you should do 2 following steps
- List all the software features (functionality, performance, GUI…) which may need to test.
- Define the target or the goal of the test based on above features
- Let's apply these steps to find the test objective of your Guru99 Bank testing project
- You can choose the 'TOP-DOWN' method to find the website's features which may need to test. In this method, you break down the application under test to component and sub-component.
- In the previous topic, you have already analyzed the requirement specs and walk through the website, so you can create a Mind-Map to find the website features as following
- Example:
- Check that whether website Guru99 functionality (Account, Deposit…) is working as expected without any error or bugs in real business environment
- Check that the external interface of the website such as UI is working as expected and & meet the customer need
- Verify the usability of the website. Are those functionalities convenient for user or not?

4. **Exit plan**

- It specifies the criteria that denote a successful completion of a test phase. The exit criteria are the targeted results of the test and are necessary before proceeding to the next phase of development. Example: 95% of all critical test cases must pass.
- Some methods of defining exit criteria are by specifying a targeted run rate and pass rate.

  1. Run rate is ratio between number test cases executed/total test cases of test specification. For example, the test specification has total 120 TCs, but the tester only executed 100 TCs, So the run rate is 100/120 = 0.83 (83%)

2. Pass rate is ratio between numbers test cases passed / test cases executed. For example, in above 100 TCs executed, there're 80 TCs that passed, so the pass rate is 80/100 = 0.8 (80%)

- This data can be retrieved in Test Metric documents.
  1. Run rate is mandatory to be 100% unless a clear reason is given.
  2. Pass rate is dependent on project scope, but achieving high pass rate is a goal.
- Test Plan Example:
  1. Your Team has already done the test executions. They report the test result to you, and they want you to confirm the Exit Criteria.
  2. In above case, the Run rate is mandatory is 100%, but the test team only completed 90% of test cases. It means the Run rate is not satisfied, so do NOT confirm the Exit Criteria

3. Test reporting plan

- What Is A Test Reporting Plan?
  1. As we know, Software Testing is an important phase in SDLC and also it serves as the "Quality Gate" for the application to pass through and certified as "Can Go Live" by the Testing Team.
  2. Test Summary Report is an important deliverable which is prepared at the end of a Testing project, or rather after Testing is completed. The prime objective of this document is to explain various details and activities about the Testing performed for the Project, to the respective stakeholders like Senior Management, Client, etc.
  3. As part of Daily Status Reports, daily testing results will be shared with involved stakeholders every day. But the Test Summary Report provides a consolidated report on the Testing performed so far for the project.
- 12 Steps Guide To Writing An Effective Test Summary Report
  1. Step #1) Purpose of the document

     <Short description of the objective of preparing the document>

     For Example, This document explains the various activities performed as part of the Testing of the 'ABCD Transport System' application.
  2. Step #2) Application Overview

     <Brief description of the application tested>

For Example, ‘ABCD Transport System’ is a web-based Bus ticket booking application. Tickets for various buses can be booked using the online facilities. Real-time passenger information is received from a ‘Central Repository System’, which will be referred before booking is confirmed. There are several modules like Registration, Booking, Payment, and Reports which are integrated to fulfill the purpose.

3. Step #3) Testing Scope

   In Scope

   Out of Scope

   Items not tested

   <This section explains the functions/modules in scope & out of scope for testing; Any items which are not tested due to any constraints/dependencies/restrictions>

   For Example, A functionality verification that needs connectivity to a third-party application cannot be tested, as the connectivity could not be established due to some technical limitations. This section should be clearly documented, else it will be assumed that Testing covered all areas of the application.

   In-Scope: Functional Testing for the following modules are in Scope of Testing

   - Registration
   - Booking
   - Payment

   Out of Scope: Performance Testing was not done for this application.

   Items not tested: Verification of connectivity with the third party system ‘Central repository system’ was not tested, as the connectivity could not be established due to some technical limitations. This can be verified during UAT (User Acceptance Testing) where the connectivity is available or can be established.

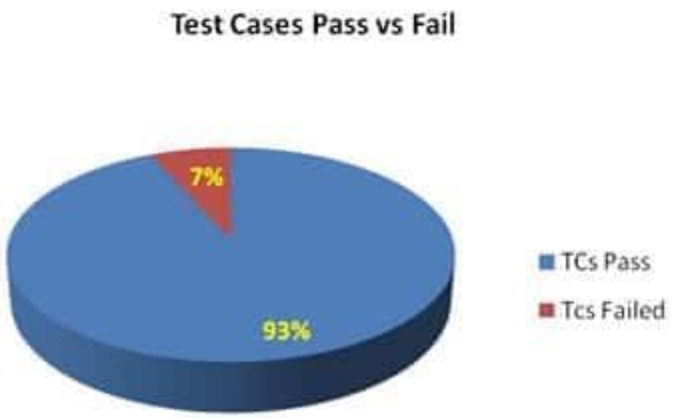
4. Step #4) Metrics

   <Metrics will help to understand the test execution results, the status of test cases & defects, etc. Required Metrics can be added as necessary. Example: Defect Summary-Severity wise; Defect Distribution-Function/Module wise; Defect Ageing etc.. Charts/Graphs can be attached for better visual representation>

   No. of test cases planned vs executed

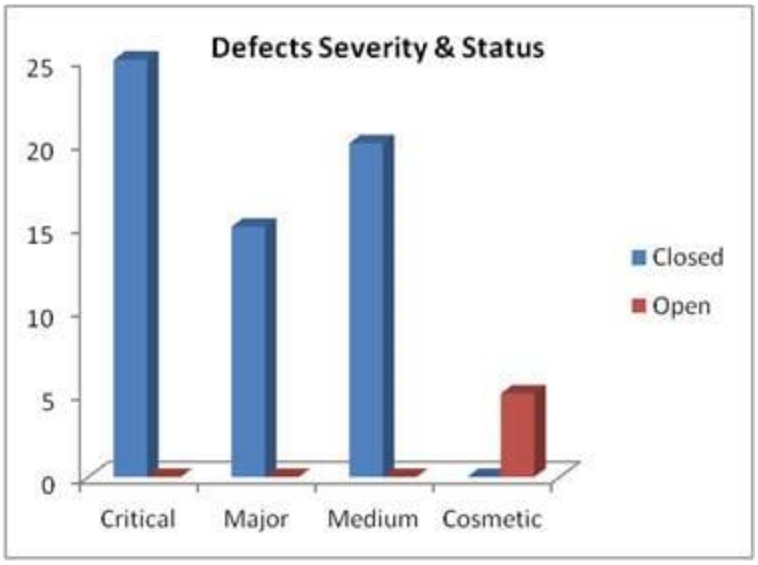
   No. of test cases passed/failed

Test cases planned	Test cases executed	TCs Pass	Tcs Failed
80	75	70	5

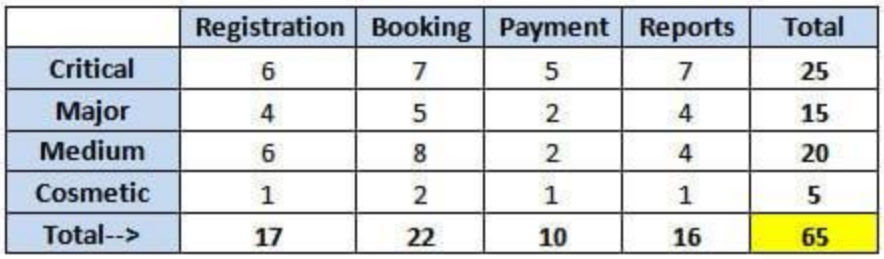


- **No of defects identified and their Status & Severity**

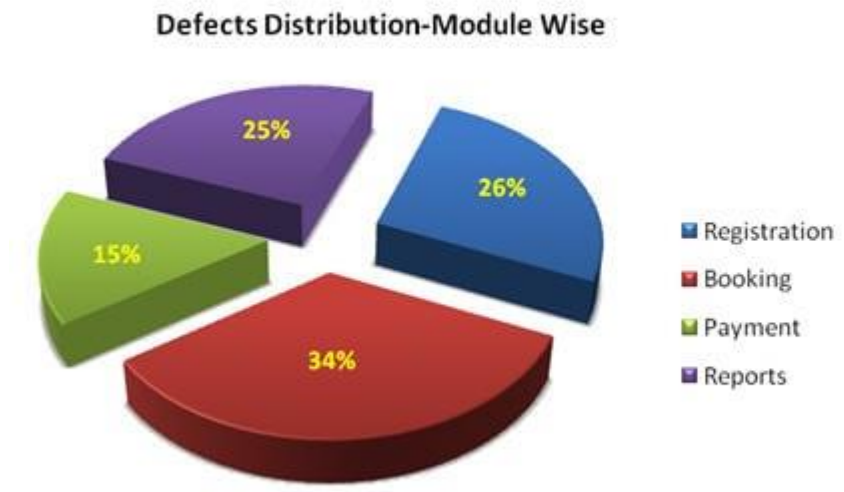
	Critical	Major	Medium	Cosmetic	Total
Closed	25	15	20	0	60
Open	0	0	0	5	5
					65



- **Defects distribution – module wise**



	Registration	Booking	Payment	Reports	Total
Critical	6	7	5	7	25
Major	4	5	2	4	15
Medium	6	8	2	4	20
Cosmetic	1	2	1	1	5
Total-->	17	22	10	16	65



5. Step #5) Types of testing performed

   Smoke Testing

   System Integration Testing

   and Regression Testing

   <Describe the various types of Testing performed for the Project. This will make sure the application is being tested properly through testing types agreed as per Test Strategy.

   Note: If several rounds of testing were done, the details can also be included here.>

   For Example,
   a) Smoke Testing

   This testing was done whenever a Build is received (deployed into Test environment) for Testing to make sure the major functionality is working fine, Build can be accepted and Testing can start.

   b) System Integration Testing

   This is the Testing performed on the Application under test, to verify the entire application works as per the requirements.

Critical Business scenarios were tested to make sure important functionality in the application works as intended without any errors.

c) Regression Testing

Regression testing was performed each time a new build is deployed for testing which contains defect fixes and new enhancements if any.

Regression Testing is being done on the entire application and not just the new functionality and Defect fixes.

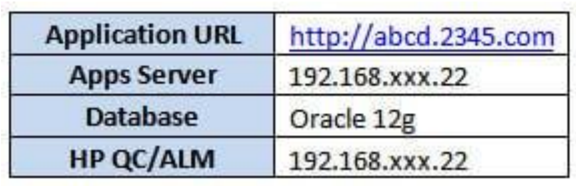
This testing ensures that existing functionality works fine after defect fix and new enhancements are added to the existing application.

Test cases for new functionality are added to the existing test cases and executed.

6. Step #6) Test Environment & Tools

<Provide details on Test Environment in which the Testing is carried out. Server, Database, Application URL, etc. If any Tools were used like Quality Center (now HP ALM) for logging defects>

For Example,

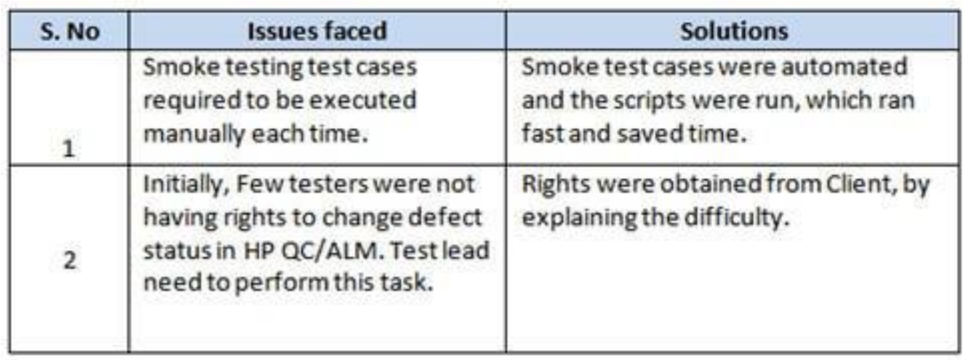


Application URL	http://abcd.2345.com
Apps Server	192.168.xxx.22
Database	Oracle 12g
HP QC/ALM	192.168.xxx.22

7. Step #7) Lessons Learned

<This section is used to describe the critical issues faced and their solutions (how they were solved during the Testing). Lessons learned will help to make proactive decisions during the next Testing engagement, by avoiding these mistakes or finding a suitable workaround>

For Example,



S. No	Issues faced	Solutions
1	Smoke testing test cases required to be executed manually each time.	Smoke test cases were automated and the scripts were run, which ran fast and saved time.
2	Initially, Few testers were not having rights to change defect status in HP QC/ALM. Test lead need to perform this task.	Rights were obtained from Client, by explaining the difficulty.

8. Step #8) Recommendations

   <Any workaround or suggestions can be mentioned here>

   For Example,

   Admin control for defect management tools can be given to Offshore Test manager for providing access to the Testing team.

   Each time the onsite Admin need not be contacted for requests whenever they arise, thereby saving time due to the geographical time zone difference.

9. Step #9) Best Practices

   <There will be a lot of activities done by the Testing team during the project. Some of them could have saved time, some proved to be a good & efficient way to work, etc. These can be documented as a 'Value Add' to showcase to the Stakeholders>

   For Example,

   A repetitive task done manually every time was time-consuming. This task was automated by creating scripts and run each time, which saved time and resources.

   Smoke test cases were automated and the scripts were run, which ran fast and saved time.

   Automation scripts were prepared to create new customers, where a lot of records need to be created for Testing.

   Business-critical scenarios are separately tested on the entire application which is vital to certify they work fine.

10. Step #10) Exit Criteria

    <Exit Criteria is defined as a Completion of Testing by fulfilling certain conditions like

(i) All planned test cases are executed;
(ii) (iI) All Critical defects are Closed etc.>

For Example,

All test cases should be executed – Yes

All defects in Critical, Major, Medium severity should be verified and closed – Yes.

Any open defects in Trivial severity – Action plan prepared with expected dates of closure.

No Severity1 defects should be 'OPEN'; Only 2 Severity2 defects should be 'OPEN'; Only 4 Severity3 defects should be 'OPEN'. Note: This may vary from project to project. Plan of Action for the Open defects should be clearly mentioned with details on when & how they will be addressed and closed.>

11. Step #11) Conclusion/Sign Off

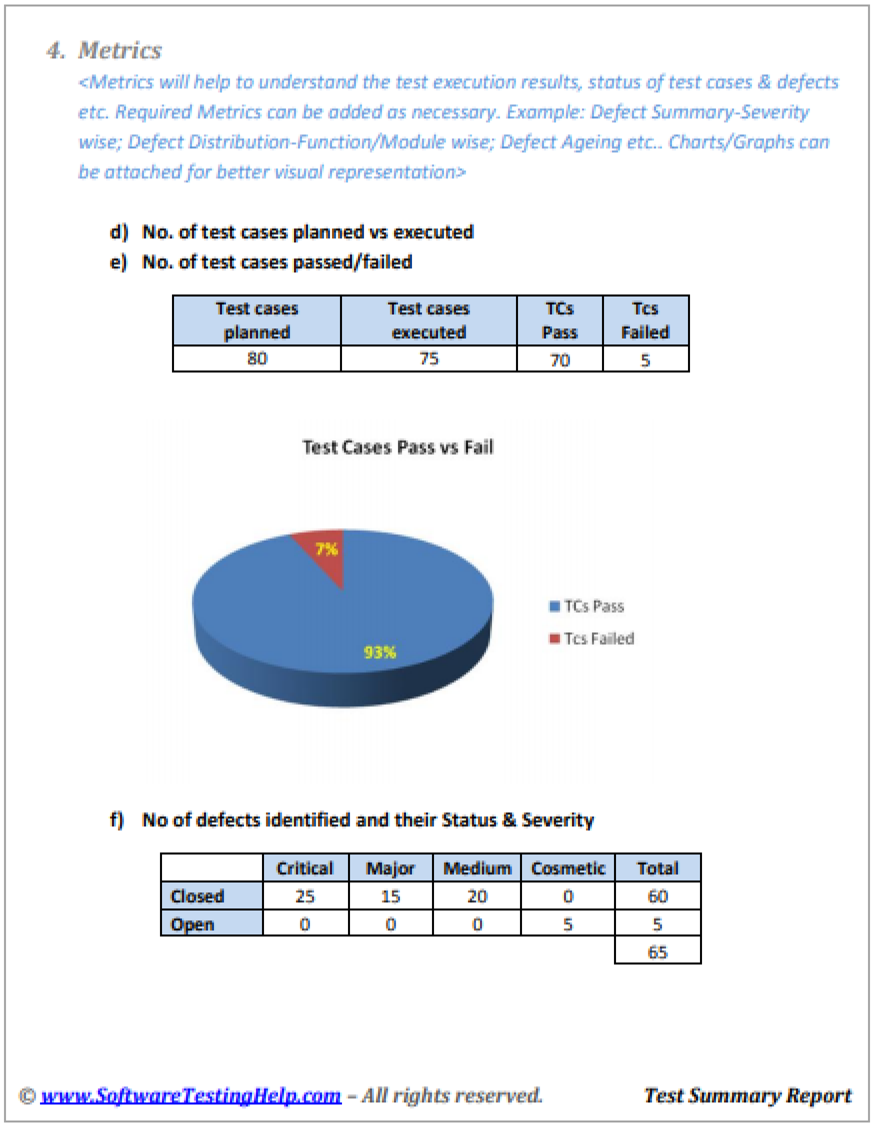
    <This section will mention whether the Testing team agrees and gives a Green signal for the application to 'Go Live' or not after the Exit Criteria was met. If the application does not meet the Exit Criteria, then it can be mentioned as – "The application is not suggested to 'Go Live'. It will be left with the decision of Senior Management and Client and other Stakeholders involved to take the call on whether the application can 'Go Live' or not.>

    For Example, As the Exit criteria were met and satisfied as mentioned in Section 10, this application is suggested to 'Go Live' by the Testing team. Appropriate User/Business acceptance testing should be performed before 'Go Live'.

12. Step #12) Definitions, Acronyms, and Abbreviations

    <This section mentions the meanings of Abbreviated terms used in this document and any other new definitions>

    => Download Sample Test Summary Report: Click here to download a sample Test Report template with an example.



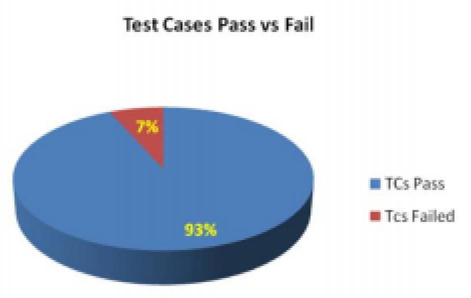
4. Metrics

<Metrics will help to understand the test execution results, status of test cases & defects etc. Required Metrics can be added as necessary. Example: Defect Summary-Severity wise; Defect Distribution-Function/Module wise; Defect Ageing etc.. Charts/Graphs can be attached for better visual representation>

d) No. of test cases planned vs executed

e) No. of test cases passed/failed

Test cases planned	Test cases executed	TCs Pass	Tcs Failed
80	75	70	5



f) No of defects identified and their Status & Severity

	Critical	Major	Medium	Cosmetic	Total
Closed	25	15	20	0	60
Open	0	0	0	5	5
					65

© www.SoftwareTestingHelp.com – All rights reserved. **Test Summary Report**

Test management ensures the delivery of high-quality products that completely suffice on customers' requirements. It assists in the delivery of software within tighter deadlines, allowing collaboration among developers and testers.

We consider M/S test manager (MTM) for the purpose of managing the test plan, recording of the test cases, and reporting the bugs.

In this lecture, the following topics are discussed:

1. Test data management

2. Test management with MTM

Any Test Data Management Tool Follows The Following Steps Of Processing:

- In any system, data is stored in different formats, types, and locations. Different rules are applied to this data. Hence, the test tool finds out the appropriate test data from these data for the testing process.
- Now the tool extracts the subset of data from the selected test data collected from multiple data sources.
- After selecting the subset test data, test tool uses masking for sensitive test data, such as a client's personal information.
- Now the tool performs the comparison between the actual data and baseline test data to check the accuracy of the application.
- To increase the efficiency of the application, the tool refreshes the test data.

3. Test planning with MTM

4. Test case development with MTM

- Test recording is the easiest way for new automation testers to start learning test automation. Identifying objects on applications is time-consuming. The Web Recorder Utility function captures your actions being performed on the application and converts them into runnable code in the back-end.
- You can quickly automate a few functionalities of your app and save time by recording actions that have to be performed many times in iterative builds. This function supports recording and running the same tests on multiple browsers.

Lesson 29

Test Execution using Microsoft (M/S) Test Manager (MTM)

Topic- 058:

We take MTM merely as an example where we consider the following aspects of test planning:

1. Test Plan development
2. Test design
3. Text execution
4. Exit plan
5. Test reporting plan

We start from a small-scale test plan for one of our projects running over the internet and are available to students as a ready reference. We execute all aspects or the stages of test management and give an entire run from planning to execution and from execution to reporting.

In this lecture, the following topics are discussed:

- Test execution with MTM
- Test execution recording with MTM

Topic- 059:

Test execution is followed by test execution reporting phase where all the potential bugs reported are discussed, this is within a reporting period and reason to use potential is that the third-hour meeting sometimes allows developers to reject findings explaining the working of the system under test.

In this lecture, the following topics are discussed:

1. Test execution reporting with MTM
2. Bug log with MTM
3. Test cycle closure with MTM

Lesson 30

Regression Testing

Topic- 060:

Regression testing Regression testing verifies if systems under evolution retain their existing functionality. Based on large test sets accumulated over time, this is a costly process, especially if testing is manual or the system to be tested is remote or only available for testing during a limited period. Often, changes made to a system are local, arising from fixing bugs or specific additions or changes to the functionality. Rerunning the entire test set in such cases is wasteful. Instead, we would like to be able to identify the parts of the system that were affected by the changes and select only those test cases for rerun which test functionality that could have been affected.

Evolution in software systems is inevitable to keep them abreast with the changing needs of businesses. To assess and assure that there is no deviation of the existing functionality, regression testing uses a comprehensive set of test cases to reevaluate every new version. Such regression test suites are accumulated over time and can be large and costly to run. In many cases, however, the impact of a particular evolution step is limited to a small part of the system, especially if maintenance is concerned with minor corrections or additions. In such cases it would be beneficial to select only those test cases for rerun which exercise parts of the system directly or indirectly affected by the changes.

In this lecture, the following topics are discussed:

- Regression testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features.
- Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
- This testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are done.

- Regression testing requirements
    - The Need of Regression Testing mainly arises whenever there is requirement to change the code and we need to test whether the modified code affects the other part of software application or not. Moreover, regression testing is needed, when a new feature is added to the software application and for defect fixing as well as performance issue fixing.

  - How changes effect system under test
  - Following are the major testing problems for doing regression testing:
  - With successive regression runs, test suites become fairly large. Due to time and budget constraints, the entire regression test suite cannot be executed
  - Minimizing the test suite while achieving maximum Test coverage remains a challenge
  - Determination of frequency of Regression Tests, i.e., after every modification or every build update or after a bunch of bug fixes, is a challenge.
- Methods in Regression Testing
  - In order to do Regression Testing process, we need to first debug the code to identify the bugs. Once the bugs are identified, required changes are made to fix it, then the regression testing is done by selecting relevant test cases from the test suite that covers both modified and affected parts of the code.
  - Software maintenance is an activity which includes enhancements, error corrections, optimization and deletion of existing features. These modifications may cause the system to work incorrectly. Therefore, Regression Testing becomes necessary. Regression Testing can be carried out using the following techniques:
- Retest All
  - This is one of the methods for Regression Testing in which all the tests in the existing test bucket or suite should be re-executed. This is very expensive as it requires huge time and resources.
- Regression Test Selection
  - Regression Test Selection is a technique in which some selected test cases from test suite are executed to test whether the modified code affects the software application or not. Test cases are categorized into two parts, reusable test cases which can be used in further regression cycles and obsolete test cases which can not be used in succeeding cycles.
- Prioritization of Test Cases
  - Prioritize the test cases depending on business impact, critical & frequently used functionalities. Selection of test cases based on priority will greatly reduce the regression test suite.
-
- Role of prioritization in regression:
  - During the test execution process, there are major challenges witnessed in regression testing as listed below :
  - Regression testing takes a huge amount of time for bigger projects.
  - In some cases, the test execution might even take one complete sprint for completion.
  - This increases the test execution time, the individual efforts along with the increased project costs.

- Also, re-executing the same set of test cases repeatedly results in lack of concentration.
- The testers might fail to test the most critical functionality of the application deriving bugs into production.
- This eventually increases the project expenditure.
- Prioritizing the test cases helps the DevOps team to reduce the cost of software test execution phase and the time taken for test execution to ensure the delivery of an exceptional quality product.
- It allows the testers to think and analyze which tests to run to manage the risks for software delivery.
- Test case prioritization offers help in detecting and correcting faults earlier than might otherwise be possible.
- Prioritization assists the team to resolve most critical defects at an early stage of a testing cycle.
- If the test cases are prioritized for regression testing, the DevTestOps team can get a chance to detect and correct the faults at an early stage as possible.

Lesson 31

Factors for regression testing

Topic- 061:

Regression testing, as discussed, is done to see if there is regression in quality after any or all of the following:

1. A requirement has evolved, i.e., it has been added, deleted, or updated
2. Bug fixing has resulted in evolution in some part of the code
3. Client has requested changes.
4. In general, any activity that results in code recompilation

Now, we need to understand that we cannot afford to neglect rerunning of test cases and we can also not rerun the whole of the test suite for petty changes. Therefore, there are two dictating aspects:

1. The nature of the application: imagine we are working on a medical application or a safety critical application, a commercial application or a class assignment. All have different testing requirements: the medical application or a safety critical application would require rerunning of all test cases even if there is a small evolution in the system. We would require to select a subset of test cases for commercial applications.
2. The effort to select a subset of test cases should not exceed effort required to rerun the whole of the test suite.
3. Rerunning of test cases has associated time and monetary costs and hence we need methods to first prioritize test cases and do test case minimization to select a subset for rerunning.

In this lecture, the following topics are discussed:

1. Effects of requirements change in regression testing
2. Effects of maintenance in regression testing
3. Impact of bug fixing in regression testing

Lesson 32

Methods in regression testing

Topic- 062:

Regression testing, as discussed, is done to see if there is regression in quality after any or all of the following:

1. A requirement has evolved, i.e., it has been added, deleted, or updated
2. Bug fixing has resulted in evolution in some part of the code
3. Client has requested changes.
4. In general, any activity that results in code recompilation

Now, we need to understand that we cannot afford to neglect rerunning of test cases and we can also not rerun the whole of the test suite for petty changes. Therefore, there are two dictating aspects:

1. The nature of the application: imagine we are working on a medical application or a safety critical application, a commercial application or a class assignment. All have different testing requirements: the medical application or a safety critical application would require rerunning of all test cases even if there is a small evolution in the system. We would require to select a subset of test cases for commercial applications.
2. The effort to select a subset of test cases should not exceed effort required to rerun the whole of the test suite.
3. Rerunning of test cases has associated time and monetary costs and hence we need methods to first prioritize test cases and do test case minimization to select a subset for rerunning.

In this lecture, the following topics are discussed:

1. Methods in regression testing
2. Role of test case prioritization in regression testing
3. Test case recording and regression testing

Lesson 33

Test Case Recording

Topic- 063:

Regression testing, as discussed, is done to see if there is regression in quality. Automated software testing can partially solve the problem of test cases rerunning, if automated.

We explain use of Microsoft Test Management software known as Microsoft Test Manager (MTM) for test planning connecting test cases to user stories and tasks. The tester can plan a test case as sequence of steps and then recording of their execution is possible. Once recorded, they can be scheduled to run in an automatic manner. We also explain use of Selenium for the same purpose. We define various application domains that can be tested using automated test management and testing tools in this way.

In this lecture, the following topics are discussed:

1. Test case recording with MTM
2. Test case recording with Selenium
3. Selenium as an Add-on for IDE

Lesson 34

Web Testing

Topic- 064:

Web testing or website testing is checking your web application or website for potential bugs before its made live and is accessible to general public. Web Testing checks for functionality, usability, security, compatibility, performance of the web application or website. During this stage issues such as that of web application security, the functioning of the site, its access to handicapped as well as regular users and its ability to handle traffic is checked.

Functionality testing in web application includes testing all links in your webpages are working correctly and make sure there are no broken links. Links to be checked will include testing of outgoing links, internal links, anchor links, mailTo links, etc. We also test business flows, positive and negative scenarios and do usability, interface, database, compatibility, performance, security and crowd testing.

In this lecture, the following topics are discussed:

1. Web testing
2. Web test recording with Microsoft Visual Studio

WEB TESTING, or website testing is checking your web application or website for potential bugs before its made live and is accessible to general public. Web Testing checks for functionality, usability, security, compatibility, performance of the web application or website.

During this stage issues such as that of web application security, the functioning of the site, its access to handicapped as well as regular users and its ability to handle traffic is checked.

Lesson 35

Load Testing

Topic- 065:

Load testing Load testing is a type of non-functional testing. A load test is type of software testing which is conducted to understand the behavior of the application under a specific expected load. Load testing is performed to determine a system's behavior under both normal and at peak conditions. Load testing is used to identify if the infrastructure used for hosting the application is sufficient. It is used to find if the performance of the application is sustainable when it is at the peak of its user load. It tells us how many simultaneous users can the application handle and the scale of the application required in terms of hardware, network capacity etc., so that more users could access the application. It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. E.g. If the number of users is increased then how much CPU, memory will be consumed, what is the network and bandwidth response time. Load testing can be done under controlled lab conditions to compare the capabilities of different systems or to accurately measure the capabilities of a single system.

In this lecture, the following topics are discussed:

1. Load Testing
2. Types of load tests

   Load Testing is a non-functional software testing process in which the performance of software application is tested under a specific expected load. It determines how the software application behaves while being accessed by multiple users simultaneously. The goal of Load Testing is to improve performance bottlenecks and to ensure stability and smooth functioning of software application before deployment.

   This testing usually identifies -

   The maximum operating capacity of an application

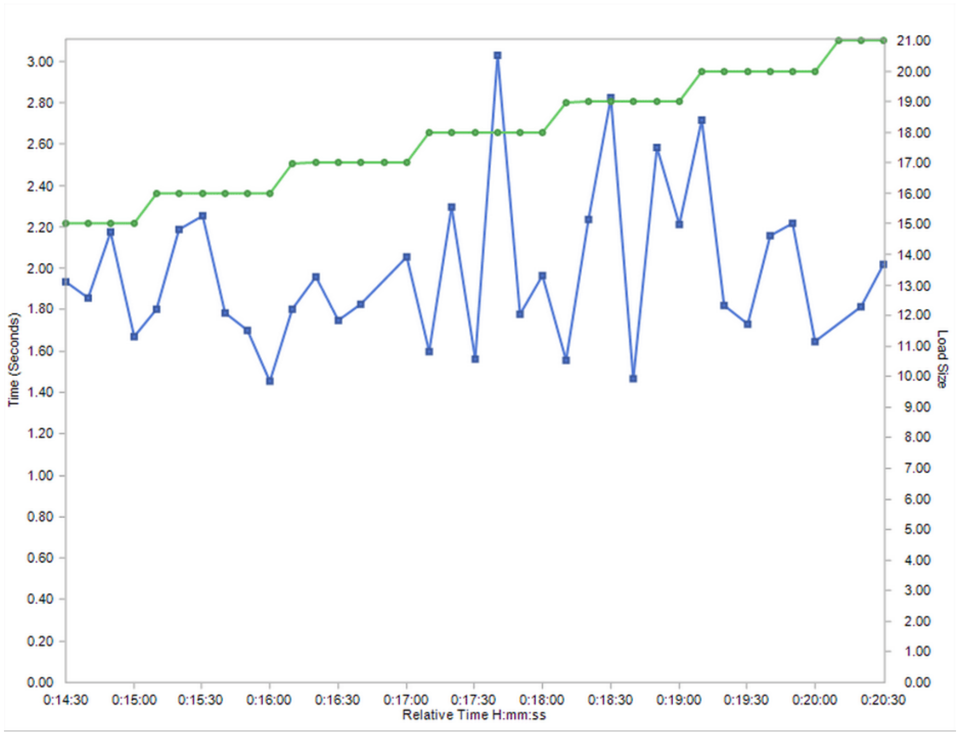
   Determine whether the current infrastructure is sufficient to run the application

   Sustainability of application with respect to peak user load

   Number of concurrent users that an application can support, and scalability to allow more users to access it.

It is a type of non-functional testing. In Software Engineering, Load testing is commonly used for the Client/Server, Web-based applications - both Intranet and Internet.

For example, in the graph below, you're running a load of 20 users to see that the page time does not exceed 3.5 seconds.



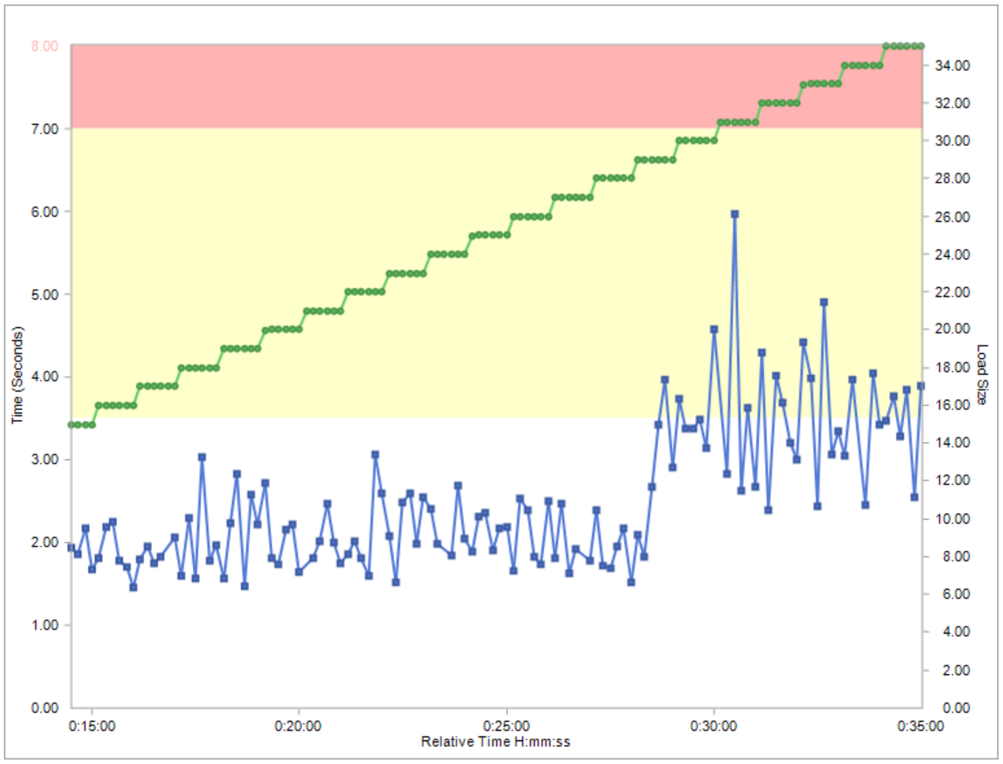
2 – Capacity Testing

Capacity testing (sometimes called scalability testing) helps you identify the maximum capacity of users the system can support, while not exceeding a max page time you defined.

The previous example showed that the system easily served 20 users with a page time of 3.5. Now you want to find out the capacity of your system – or at which point it will not be able to keep the 3.5 seconds response time? Will it be at 21 users, 30, 40, or 50 users?

Your higher-level objective is to identify the 'safety zone' of your system. To what extent can you 'stretch' it, without hurting end user experience?

In the results graph that follows you'll notice that for a page time of 3.5 seconds, the system supports 28 users. But when the load size reaches 29, page time starts exceeding the threshold of 3.5 seconds.



Topic- 066:

Load testing involves simulating real-life user load for the target application. It helps you determine how your application behaves when multiple users hit it simultaneously. Load testing differs from stress testing, which evaluates the extent to which a system keeps working when subjected to extreme workloads or when some of its hardware or software has been compromised. The primary goal of load testing is to define the maximum amount of work a system can handle without significant performance degradation. Load testing falls under the category – non-functional testing. It's mainly used for testing the performance of Client/Server and applications which are web based. In many organizations load testing is performed at the end of the software development life cycle while some organization do not perform load testing at all. In case there are performance issues in the application, this could result in loss of revenue to the customer.

In this lecture, the following topics are discussed:

1. Load Testing

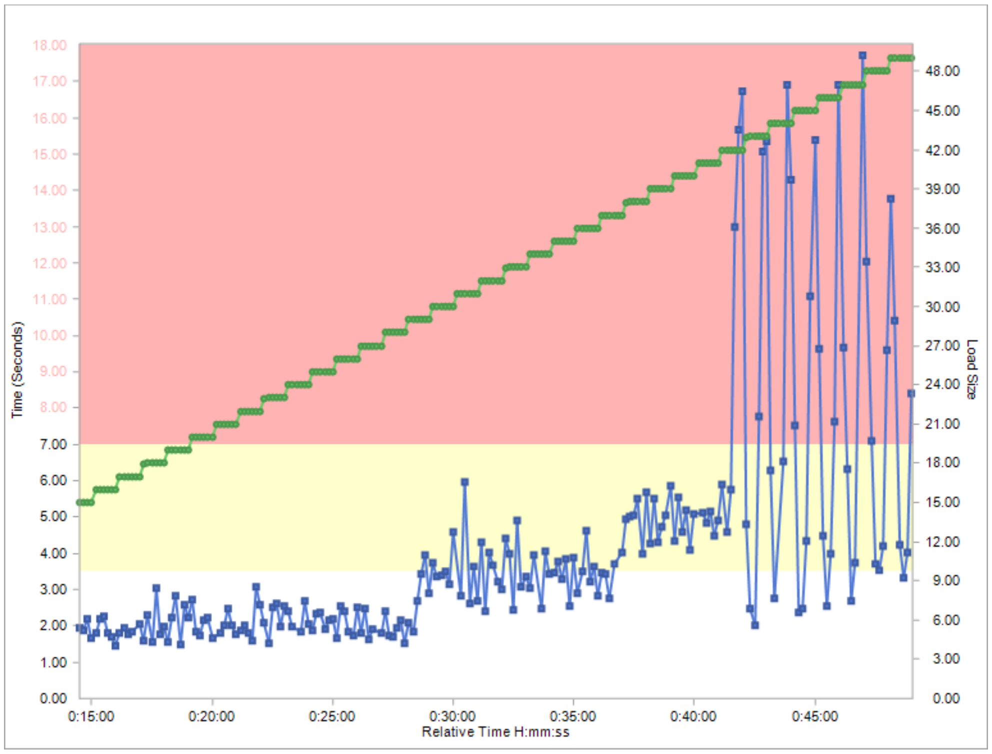
2. Types of load tests
3. Load / stress / performance testing

3.1 – Stress Testing

The objective of a stress test is to find how a system behaves in extreme conditions. You purposely try to break your system, using any set of extreme conditions – whether doubling the number of users, using a database server with much less memory, or a server with a weaker CPU.

What you're trying to find out is how will the system behave under stress and what will be the user experience? Will the system start throwing out errors? Will response time double? Or will the entire system get stuck and crash?

To continue with the previous example, instead of stopping at around 30 users (when the page time exceeds your goal), you'd continue to increase the user load on the system. Your stress test discovers that up to a load size of 41 users, the system functions, despite the increase in page time. But when the load size reaches 42 users, the system start to deteriorate, with page time reaching 15-17 seconds.



3.2 – Soak Testing

Quite often, the ‘standard’ load testing, which is run for a short limited time will not succeed uncovering all problems. A production system typically runs for days, weeks and months. For example, an eCommerce application must be available 24×7 and a stock exchange application will run continuously on work days. The duration of a soak test should have some correlation to the operational mode of the system under test.

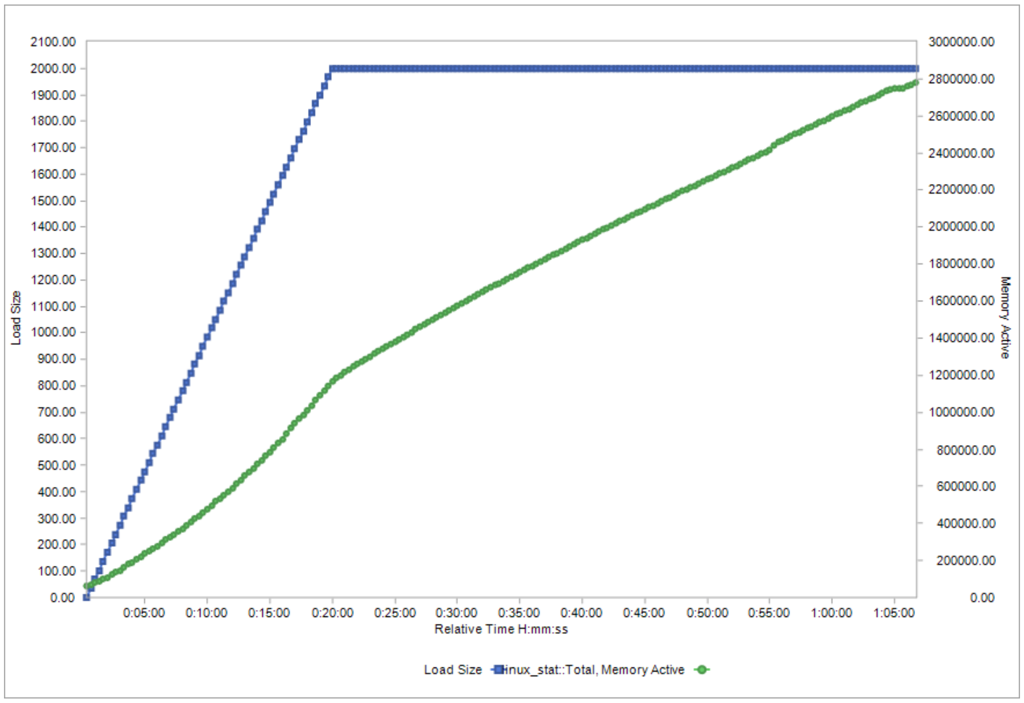
Soak testing aims to uncover performance problems that surface only during a long duration of time. During soak testing, you’re asking questions such as:

Is there a constant degradation in response time when the system is run over time?

Are there any degradations in system resources that are not apparent during short runs, but will surface when a test is run for a longer time? For example, memory consumptions, free disk space, machines handles, etc.

Is there any periodical process that may affect the performance of the system, but which can only be detected during a prolonged system run? For example, a backup process that runs once a day, exporting of data into a 3rd party system, etc.

When running soak testing, you’re looking for trends and changes in system behavior. In the following example, notice how the load size is constant for a while, but still, there is a memory leak (=green line). At first this memory leak may not affect the system, but after a while the system may crash. (The example below is shorter than a typical soak test, which would last hours, days and even weeks.)



3.3 Performance testing

Performance testing is a type of testing for determining the speed of a computer, network or device. It checks the performance of the components of a system by passing different parameters in different load scenarios.

Load testing

Load testing is the process that simulates actual user load on any application or website. It checks how the application behaves during normal and high loads. This type of testing is applied when a development project nears to its completion.

3.4 Stress testing

Stress testing is a type of testing that determines the stability and robustness of the system. It is a non-functional testing technique. This testing technique uses auto-generated simulation model that checks all the hypothetical scenarios.

4. Load testing with M/S Visual Studio

Lesson 36

Software Testing Metrics

Topic- 067:

We can increase testing efficiency by setting quality goals with the right set of *software quality metrics* to track them. Collecting and documenting test cases is a good start, but if we do not set goals it is just too easy to lose focus during the day-to-day activities. Software quality metrics help to stay on track during a testing project and measure progress.

By looking at a chart of passed and failed tests over time, one can quickly judge if quality level is increasing towards the release date or if we need to take action. There are several types of metrics that need to be collected and analyzed that indicate various aspects of software quality of our system under test.

In this lecture, the following topics are discussed:

1. Software test metrics
2. Importance of test metrics

Topic- 068:

Broadly speaking, software testing metrics can be characterized as Process Metrics which can be used to improve the process efficiency of the SDLC (Software Development Life Cycle), Product Metrics which deal with the quality of the software product and Project Metrics that can be used to measure the efficiency of a project team or any testing tools being used by the team members.

In this lecture, the following topics are discussed:

1. Test metrics classifications
2. Test metrics lifecycle

Lesson 37

Test Metrics Classification

Topic- 069:

There are two types of metrics, basic metrics and calculated metrics. Base metrics is the raw data collected by Test Analyst during the test case development and execution (# of test cases executed, # of test cases). While calculated metrics are derived from the data collected in base metrics. Calculated metrics is usually followed by the test manager for test reporting purpose (% Complete, % Test Coverage). Depending on the project or business model some of the important metrics are test case execution productivity metrics, test case preparation productivity metrics, defect metrics, defects by priority, defects by severity, and defect slippage ratio. All of these are explained in the presentation slides with division relative to defect and testing efficiency.

In this lecture, the following topics are discussed:

1. Test metrics classification
2. Defect related metrics
3. Testing efficiency metrics

Topic- 070:

Performance testing is a form of software testing that focuses on how a system running the system performs under a particular load. This is not about finding software bugs or defects. Performance testing measures according to benchmarks and standards. Performance testing should give developers the diagnostic information they need to eliminate bottlenecks.

To understand how software will perform on users' systems, there different types of performance tests that can be applied during software testing. This is non-functional testing, which is designed to determine the readiness of a system. (Functional testing focuses on individual functions of software.). Performance testing metrics are related to system performance where this metric determines the product quality-based performance criteria on which one can take decision for releasing of the product to next phase i.e. it indicates quality of product under test with respect to performance. If requirement is not met, one can assign the severity for the requirement so that decision can be taken for the product release with respect to performance.

In this lecture, the following topics are discussed:

1. Performance testing metrics
2. Performance testing efficiency metrics
3. Automation testing metrics
4. Common testing metrics

Lesson 38

Penetration Testing

Topic- 071:

Penetration testing, also called pen testing or ethical hacking, is the practice of testing a computer system, network or web application to find security vulnerabilities that an attacker could exploit. Penetration testing can be automated with software applications or performed manually. Either way, the process involves gathering information about the target before the test, identifying possible entry points, attempting to break in -- either virtually or for real -- and reporting back the findings. Objective of penetration testing is to identify security weaknesses.

In this lecture, the following topics are discussed:

1. Penetration Testing
2. Penetration Testing phases

Topic- 072:

Penetration testing can also be used to test organization's security policy, its adherence to compliance requirements, its employees' security awareness and the organization's ability to identify and respond to security incidents. Typically, the information about security weaknesses that are identified or exploited through pen testing is aggregated and provided to the organization's IT and network system managers, enabling them to make strategic decisions and prioritize remediation efforts. Penetration tests are also sometimes called white hat attacks because in a pen test, the good guys are attempting to break in. This lecture covers penetration testing introduction, phases, types and tools.

In this lecture, the following topics are discussed:

1. Types of penetration testing
2. Penetration Testing tools

Topic- 073:

We discuss how penetration testing is different from hacking and why it is called ethical hacking. We end up our discussion with an example.

In this lecture, the following topics are discussed:

1. Penetration Testing or hacking
2. Penetration Testing – example

Lesson 39

Exploratory Testing

Topic- 074:

Exploratory testing is, more than strictly speaking a “practice,” a style or approach to testing software which is often contrasted to “scripted testing”. It emphasizes the tester’s autonomy, skill and creativity, much as other Agile practices emphasize these qualities in developers. Exploratory testing recommends performing various test-related activities (such as test design, test execution, and interpretation of results) in an interleaved manner, throughout the project, rather than in a fixed sequence and at a particular “phase”. Exploratory testing also emphasizes the mutually supportive nature of these techniques, and the need for a plurality of testing approaches rather than a formal “test plan”.

In this lecture, the following topics are discussed:

1. Exploratory Testing
2. Exploratory Testing phases

Topic- 075:

We explain how exploratory testing can be used to do testing of commercial applications, what are its phases, and how can be make use of software such as MTM to undertake this form of testing.

In this lecture, the following topics are discussed:

1. Exploratory Testing vs structural testing
2. Exploratory Testing benefits

Topic- 076:

We explain exploratory testing with the help of a worked example.

In this lecture, the following topics are discussed:

1. Exploratory Testing: worked example

Lesson 40

Usability and Documentation Testing

Topic- 077:

Usability testing is a way to see how easy to use something is by testing it with real users. Users are asked to complete tasks, typically while they are being observed by a researcher, to see where they encounter problems and experience confusion. If more people encounter similar problems, recommendations will be made to overcome these usability issues.

Usability testing is a method used to evaluate how easy a website is to use. The tests take place with real users to measure how 'usable' or 'intuitive' a website is and how easy it is for users to reach their goals. Before a new product is released, explorative usability testing can establish what content and functionality a new product should include to meet the needs of its users. Users test a range of different services where they are given realistic scenarios to complete which helps to highlight any gaps in the market that can be taken advantage of and illustrate where to focus design effort.

In this lecture, the following topics are discussed:

1. Usability Testing
2. Usability Testing Process
3. Methods of Usability Testing

Topic- 078:

Usability testing is an effort to ascertain the degree to which software has met usability needs of intended user base. Usability Testing is an attempt to quantify software user-friendliness according to skill needed to learn the software, time required to become efficient in using software, the measured increase in user productivity and a subjective assessment of a user's attitude toward using the software. We explain our standpoint with the help of a worked example.

In this lecture, the following topics are discussed:

1. Benefits and drawbacks of usability testing
2. Usability testing – worked example

Topic- 079:

Software documentation, some 25 years ago was merely a readme.txt file copied onto the software's floppy disk. It was usually a 1-page insert put into the shrink-wrapped package containing the software and the comments used to be in the source code. Testers used to run a spell checker on the file and that was about the extent of testing documentation. Much of the non-code was the software documentation, which requires much effort to produce.

Documentation is now a major part of a software system and It may exceed the amount of source code. Currently, it might be integrated into the software (e.g., help system) and testers have to cover the code and the documentation. It is pertinent to note that assuring documentation is correct is part of a software tester's job.

In this lecture, the following topics are discussed:

1. Documentation Testing
2. Aspects of documentation testing

Lesson 41

GUI Testing

Topic- 080:

Graphical user interface (GUI) testing is a software testing type that checks the Graphical User Interface of the Application Under Test. GUI testing involves checking the screens with the controls like menus, buttons, icons, and all types of bars - toolbar, menu bar, dialog boxes, and windows, etc. The purpose of Graphical User Interface (GUI) Testing is to ensure UI functionality works as per the specification. GUI is what the user sees. A user does not see the source code. The interface is visible to the user. Especially the focus is on the design structure, images that they are working properly or not.

In this lecture, the following topics are discussed:

1. Graphical user interface (GUI) testing
2. GUI testing challenges
3. GUI testing methods

Topic- 081:

While doing GUI testing, we first fix our focus saying if we are treating GUI as a stimulus to execute functionality and therefore, we try to see if a particular use of a system is executable from a given scenario. Considering it non-functional aspect, we check all the GUI elements for size, position, width, length, and acceptance of characters or numbers. For instance, one must be able to provide inputs to the input fields. We also check if we can execute the intended functionality of the application using the GUI. We consider Error Messages are displayed correctly and clear demarcation of different sections on screen. We evaluate if font used in an application is readable, alignment of the text, color of the font and warning messages is aesthetically pleasing and uniform, images have good clarity and if they are aligned and especially focus on the positioning of GUI elements for different screen resolution. Our fixing of focus leads us towards GUI test coverage and we provide a worked example to explain it.

In this lecture, the following topics are discussed:

1. GUI testing Coverage
2. GUI testing – worked example

Lesson 42

Acceptance Testing

Topic- 082:

Acceptance is a level of software testing where a system is tested for acceptability. The purpose of this test is to evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery. Acceptance testing is testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. It is usually a Black Box Testing method and it does not normally follow a strict procedure and is not scripted but is rather ad-hoc. There are several forms and stages of acceptance testing. Internal Acceptance Testing (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support. External Acceptance Testing is performed by people who are not employees of the organization that developed the software. Customer Acceptance Testing is performed by the customers of the organization that developed the software. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.] User Acceptance Testing (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers’ customers.

In this lecture, the following topics are discussed:

1. Acceptance testing
2. Acceptance testing phases
3. Acceptance testing exit criteria
4. Acceptance testing activities

Lesson 43

Reliability Testing

Topic- 083:

Reliability testing is a software testing type, that checks whether the software can perform a failure-free operation for a specified period of time in a particular environment. Reliability means "yielding the same," in other terms, the word "reliable" means something is dependable and that it will give the same outcome every time. The same is true for Reliability testing. Reliability testing in software assures that the product is fault free and is reliable for its intended purpose. Factors influencing reliability testing includes number of faults presents in the software and the way users operate the system.

Reliability Testing is a key to better software quality. This testing helps discover many problems in the software design and functionality. The main purpose of reliability testing is to check whether the software meets the requirement of customer's reliability. Reliability testing will be performed at several levels. Complex systems will be tested at unit, assembly, subsystem and system levels.

In this lecture, the following topics are discussed:

1. Reliability testing
2. Reliability testing characteristics
3. Reliability testing types

Topic- 084:

Reliability testing is done to test the software performance under the given conditions and the objective behind performing reliability testing are to find the structure of repeating failures, to find the number of failures occurring is the specified amount of time, to discover the main cause of failure, and to conduct Performance Testing of various modules of software application after fixing defect.

After the release of the product too, we can minimize the possibility of occurrence of defects and thereby improve the software reliability. Some of the tools useful for this are- Trend Analysis, Orthogonal Defect Classification and formal methods, etc. Software reliability testing includes Feature Testing, Load Testing and Regression Testing

In this lecture, the following topics are discussed:

1. Models of software reliability
2. Software Reliability metrics

Lesson 44

Software Testing for Cleanroom Software Engineering

Topic- 085:

It is an engineering approach which is used to build correctness in developed software and the main concept behind the cleanroom software engineering is to remove the dependency on the costly processes.

The cleanroom software engineering includes the quality approach of writing the code from the beginning of the system and finally gathers into a complete a system. Following tasks occur in cleanroom engineering, incremental planning where an incremental plan is developed. The functionality of each increment, projected size of the increment and the cleanroom development schedule is created. The care is to be taken that each increment is certified and integrated in proper time according to the plan. Next is requirements gathering which is done using the traditional techniques like analysis, design, code, test and debug and a more detailed description of the customer level requirement is developed. We then develop box structure specification which is used to describe the functional specification. The box structure separates and isolates the behavior, data and procedure in each increment. Next, we develop a formal design which is a natural specification by using the black box structure approach. The specification is called as state boxes and the component level diagram called as the clear boxes. We develop correctness verification on the design and then the code. Verification starts with the highest-level testing box structure and then moves toward the design detail and code.

We then do code generation, inspection and verification followed by statistical test planning. Our main aim is to study this phase as part of this course. Here, we analyze, plan and design the projected usages of the software and we then do statistical use testing which is exhaustive testing of computer software is impossible. It is compulsory to design limited number of test cases. Statistical use technique executes a set of tests derived from a statistical sample in all possible program executions. These samples are collected from the users from a targeted population. After the verification, inspection and correctness of all errors, the increments are certified and ready for integration.

In this lecture, the following topics are discussed:

1. Cleanroom software engineering
2. Cleanroom software Eng. principles
3. Cleanroom software Engineering process

Topic- 086:

Cleanroom process model uses a method called box structure specification where a 'box' contains the system or the aspect of the system in detail. The information in each box specification is sufficient to define its refinement without depending on the implementation of other boxes. The cleanroom process model uses three types of boxes namely black-box, state-box and glass-box

The black box identifies the behavior of a system. The system responds to specific events by applying the set of transition rules. State box consists of state data or operations that are similar to the objects. The state box represents the history of the black box i.e., the data contained in the state box must be maintained in all transitions. Finally, clear-box represents transition function used by the state box is defined in the clear box. It simply states that a clear box includes the procedural design for the state box.

In this lecture, the following topics are discussed:

1. Cleanroom software engineering benefits
2. Cleanroom software Eng. – box principles
3. Cleanroom software Eng. – testing focus

Lesson 45

Program Representations for testing and Conclusion

Topic- 087:

This is the final topic of the course where only consider a couple of topics related to unit testing using Java and Junit is explained, various program representations are discussed to close the loop and a conclusion of the course with a short question-answer session is planned.

As a step towards engaging students towards latest research and techniques development for manual and automated testing. Finally, a couple of slides are dedicated to conclusion where students can ask if they feel any deficiency.

In this lecture, the following topics are discussed:

1. Software testing support for Java related IDE
2. Program Representations for Testing
3. Software Verification and Validation – Conclusion

References:

- Paul Jorgensen, (2015), Software Testing, A Craftsman's Approach, Fourth Ed. CRC Press, Taylor and Francis Group
- Bernard Homes, (2012). Fundamentals of Software Testing, ISTE, Wiley Publishers
- "Ian Sommerville, Software Engineering, ninth edition, Addison Wesley, 2009
- Glenford J. Myers, Corey Sandler, Tom Badgett. The Art of Software Testing, 3rd Edition, McGraw Hill,
- Specification based testing, available at "https://www.vskills.in/certification/tutorial/software-testing/specification-based-testing/", accessed on

- Ron Patton (2005), Software Testing, 2nd Edition, 2005, SAMS Publishers
- Paul C. Jorgensen, Software Testing: A Craftsman's Approach, Fourth Edition, CRC Press,
- Use case based testing, available at https://www.softwaretestinghelp.com/use-case-testing/ , accessed on 5/12/19
- Model-based testing, available at https://www.guru99.com/model-based-testing-tutorial.html, accessed on 5/12/19
- Jeff Tian (2005), Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement February 2005 Wiley-Inter science, 605 Third Avenue New York, NY United States, ISBN:978-0-471-71345-6
- Fault-based testing, available at http://www.cs.toronto.edu/~chechik/courses18/csc410/Ch16-FaultBasedTesting.pdf, accessed on 5/12/19
- Glenford J. Myers and Corey Sandler. 2004. The Art of Software Testing. John Wiley & Sons, Inc., Hoboken, NJ, USA.
- Test management, available at https://www.testbytes.net/blog/test-management-process/, accessed on 5/12/19
- Lee Copeland. 2003. A Practitioner's Guide to Software Test Design. Artech House, Inc., USA.
- Penetration testing, available at https://searchsecurity.techtarget.com/definition/penetration-testing, accessed on 5/12/19

- Mark Fewster and Dorothy Graham, Software Test Automation – Effective Use of Test Execution Tools, Addison Wesley
- acceptance testing, available at http://softwaretestingfundamentals.com/acceptance-testing/, accessed on 5/12/19
- Bernard Homes, (2012). Fundamentals of Software Testing, ISTE, Wiley Publishers
- Singh, Y. (2011). Software Testing. Cambridge: Cambridge University Press. doi:10.1017/CBO9781139196185