Testing Concepts

Lesson 1: Fundamentals of Testing

Lesson Objectives

To understand the following topics:

- Some Facts about Software Systems
- What is Testing?
  - Typical Objectives of Testing
  - Testing and Debugging
- Why is Testing Necessary?
  - Testing's Contributions to Success
  - Quality Assurance and Testing
  - Errors, Defects, and Failures - Reasons behind Errors
  - Defects, Root Causes and Effects
  - Cost of Software Defects
  - Importance of Testing Early in SDLC phases

Lesson Objectives

- Seven Testing Principles
  - Economic of Testing
  - Scope of Software Testing
  - Factors influencing Software Testing
- Test Process
  - Test Process in Context
  - Test Activities and Tasks
  - Test Work Products
  - Traceability between the Test Basis and Test Work Products
- The Psychology of Testing
  - Human Psychology and Testing
    - Attributes of a good Tester
    - Code of Ethics for Tester
  - Tester's and Developer's Mindsets
- Limitations of Software Testing

Some Facts about Software Systems !!

- Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars).
- Most people have had an experience with software that did not work as expected.
- Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death.

Examples :

1. Excel gives 77.1 x 850 = 100000 instead of 65535
2. Y2K problem in Payroll systems designed in 1974
3. Disney's Lion King – Simba

1.1 What is Testing ?

Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

- Misperceptions of Testing :
  - Testing only consists of running tests i.e. executing the software and checking the results. But, software testing is a process which includes many different activities and test execution is just one of these activities. The test process includes activities such as test planning, analyzing, designing, implementing and executing the tests, reporting test progress and results, and evaluating the quality of a test object.
  - Testing focuses entirely on verification of requirements, user stories, or other specifications. But, testing also involves checking whether the system meets specified requirements, it also involves validation, which is checking whether the system will meet user and other stakeholder needs in its operational environment(s).

1.1 What is Testing ? (Cont.)

Testing involves both Dynamic testing and Static testing.

- Testing that involves the execution of the component or system being tested; such testing is called **dynamic testing.**
- Testing that involves the reviewing of work products such as requirements, user stories, and source code; such testing is called **static testing.**

Test activities are organized and carried out differently in different lifecycles, explained in lesson 2.

1.1.1 Objectives of Software Testing

- To evaluate work products such as requirements, user stories, design, and code
- To verify whether all specified requirements have been fulfilled
- To validate whether the test object is complete and works as the users and other stakeholders expect
- To build confidence in the level of quality of the test object
- To prevent defects
- To find failures and defects in the Software before User finds it
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object.

1.1.1 Objectives of Software Testing (Cont.)

- To reduce the level of risk of inadequate software quality (e.g., previously undetected failures occurring in operation) and contribute to the delivery of higher quality software product

- To comply with contractual, legal, or regulatory requirements or standards, and/or to verify the test object's compliance with such requirements or standards

Objectives of testing can vary, depending upon the context of the component or system being tested, the test level, and the software development lifecycle model.

Software Testing - Definitions

The process of executing a program (or part of a program) with the intention of finding errors - G. J. Myers

Software testing is the process of testing the functionality and correctness of software by running it

Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements - IEEE 83a

The process of analyzing a system to detect the difference between existing and required conditions and to evaluate the feature of the system - IEEE/ANSI, 1983 [Std 829-1983]

1.1.2 Testing and Debugging

Testing and debugging are different.

- Executing tests can show failures that are caused by defects in the software.
- Debugging is the development activity that finds, analyzes, and fixes such defects.

Subsequent confirmation testing checks whether the fixes have resolved the defects.

In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging and associated component testing.

However, in Agile development and in some other lifecycles, testers may be involved in debugging and component testing.

1.2 Why is Software Testing necessary ?

- Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation.
- When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems.
- Software testing is also required to meet contractual or legal requirements or industry-specific standards.
- SDLC consists of many stages and if bugs are caught in the earlier stages it costs much less to fix them. This saves money and time.
- Software testing provides product security – the user gets a trustworthy product.
- Customer satisfaction - Software Testing helps in bringing out the best user experience possible.

1.2.1 Testing's Contributions to Success

Use of appropriate test techniques applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the SDLC can reduce the frequency of problematic deliveries.

Examples :

- Testers involved in requirements reviews reduces the risk of incorrect functionality being developed.
- Testers working closely with system designers reduces the risk of fundamental design defects.
- Testers working closely with developers reduces the risk of defects within the code and the tests.
- Testers verifying and validating the software prior to release can detect failures that increases the likelihood that the software meets stakeholder needs and satisfies requirements.
- Achievement of defined test objectives contributes to the success of overall software development and maintenance.

1.2.2 Quality Assurance and Testing

- Quality Assurance (QA) and Testing are not the same, but they are related - Quality management, ties them together.

- Quality management includes both **quality assurance** and **quality control.**

1.2.2 Quality Assurance and Testing (Cont.)

Quality Assurance	Quality Control
It is an preventive approach – prevents the faults from occurring by providing rules and methods.	It is a corrective approach – corrects the faults when they occur.
It is a task conducted in the process.	It is a task conducted on the product.
Gives confidence to customer.	Gives confidence to producer.
It is a set of planned and systematic set of activities that provides adequate confidence and assures that the product conforms to specified requirements.	It is the process by which product quality is compared with applicable standards and appropriate action is taken when non-conformance is detected.
Entire team (designers, coders, testers, etc.) is responsible for QA.	Only tester is responsible for QC.
Examples : use of CASE (engineering) and CAST (testing) tools, training and	Examples: Walkthrough, inspections, etc.

1.2.3 Errors, Defects and Failures

- A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product.
- An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product.

Example :

- A requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code. If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances. For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

1.2.3 Errors, Defects and Failures (Cont.)

- Error(Mistake): A human action that produces an incorrect result
- Fault: A stage caused by an error which leads to unintended functionality of the program
- Bug: It is an evidence of the fault. It causes the program to perform in unintended manner. It is found before application goes into beta version
- Failure: Inability of the system to perform functionality according to its requirement
- Defect: It is a mismatch of the actual and expected result identified while testing the software in the beta version

Reasons behind Errors

- Time pressure
- Human fallibility
- Inexperienced or insufficiently skilled project participants
- Miscommunication between project participants, including miscommunication about requirements and design
- Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used
- Misunderstandings about intra-system and inter-system interfaces, especially when such intra-system and inter-system interactions are large in number
- New, unfamiliar technologies
- Environmental conditions - for example, radiation, electromagnetic fields, and pollution can cause defects in firmware or influence the execution of software by changing hardware conditions.

1.2.4 Defects, Root Causes and Effects

- The root causes of defects are the earliest actions or conditions that contributed to creating the defects.
- Defects can be analyzed to identify their root causes.
- By focusing on most significant root causes, root cause analysis can lead to process improvements that prevent future defects from being introduced.

Example :

- Suppose incorrect interest payments, due to a single line of incorrect code, results in customer complaints. The defective code was written for a user story which was ambiguous, due to the product owner's misunderstanding of how to calculate interest. Therefore, it means that :
  - customer complaints – **effects**
  - incorrect interest payments – **failures**
  - improper calculation in the code – **defect**
  - lack of knowledge on the part of the product owner - **root cause** of this defect
  - Due to lack of knowledge, Product owner makes a **mistake** while writing user story.

Cost of Software Defects

It is Easy to find and fix defect in early stages rather than in the later phases of software.

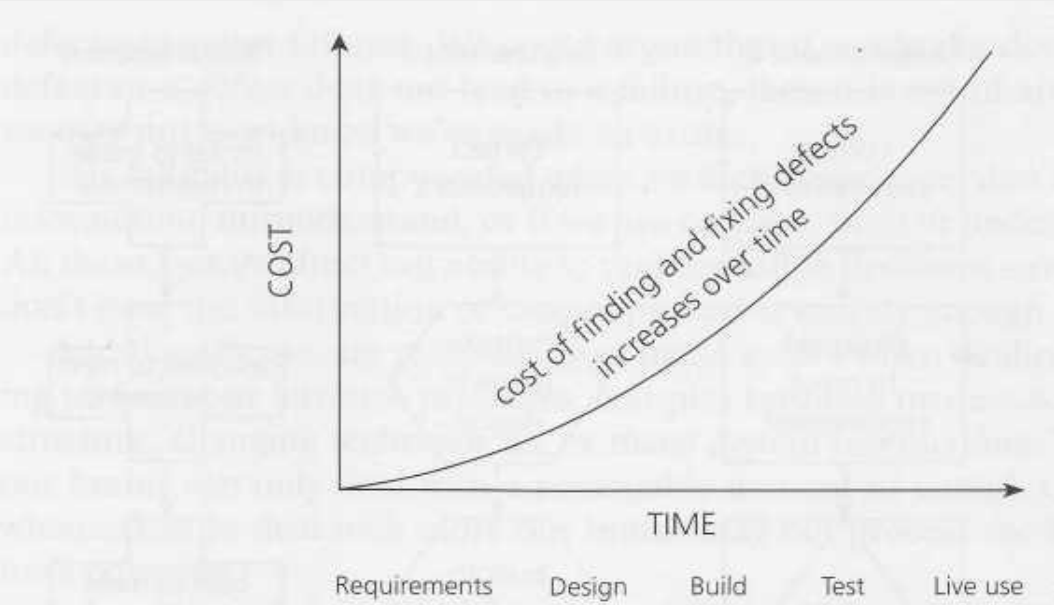


FIGURE 1.2 Cost of defects

Importance of Testing Early in SDLC Phases

- Prevents future Problems thus lowering the cost
- Testing will not be a bottleneck anymore
- Testers become more familiar with the software, as they are more involved with the evolution of the product.
- Reduces the chances of failure
- The test environment can be prepared in advance
- The risk of having a short time for testing is greatly reduced
- Maintains “quality culture” in the organization

1.3 Seven Testing Principles

Principle 1 - Testing shows presence of defects, not their absence

Principle 2 - Exhaustive testing is impossible

Principle 3 - Early testing saves time and money

Principle 4 - Defects Cluster together

Principle 5 – Beware of the Pesticide Paradox

Principle 6 - Testing is context dependent

Principle 7 - Absence of Errors is fallacy

Economics of Testing

Economics of Testing

- It is both the driving force and the limiting factor

Driving - Earlier the errors are discovered and removed in the lifecycle, lowers the cost of their removal.

Limiting - Testing must end when the economic returns cease to make it worth while i.e. the costs of testing process significantly outweigh the returns

Scope of Software Testing

Bad news : You can't test everything

Good news : There is such a thing as "good enough"

Bad news : Good enough may cost too much

What do we do : Increase focus via systematic process of elimination

What you might test?

- Those areas which are within the scope of your project

What you should test?

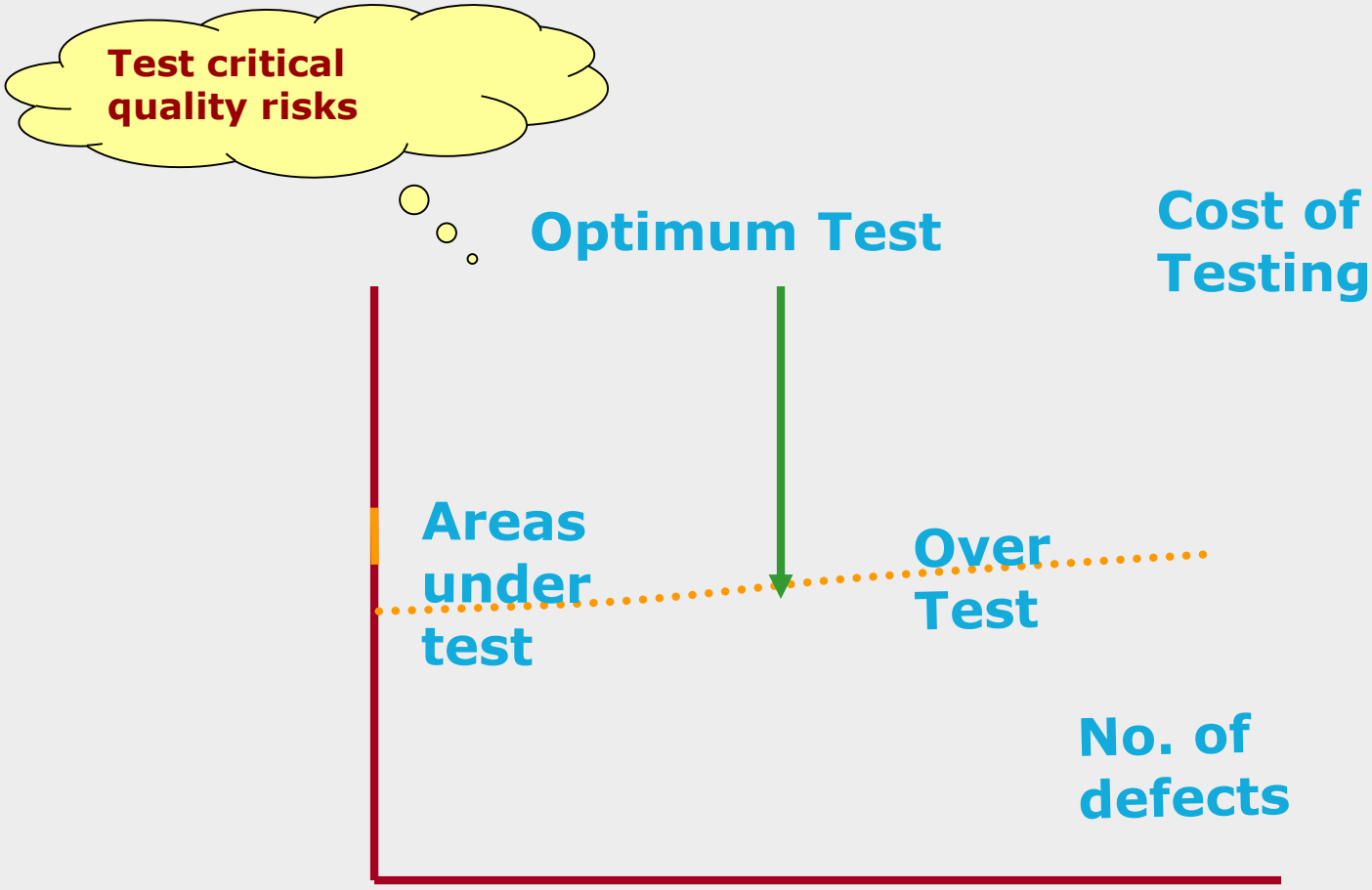
- The critical system functionality which effects the customers & users experience of quality

What you can test?

- Estimate time & resource for risk driven effort

Scope of Software Testing

"Understanding risk is the key to Optimum testing"



Factors influencing the Scope of Testing

Contractual requirements

Legal requirements

- Privacy related laws
- Non-disclosure of identity

Industry-specific requirements

- Aircraft safety equipment

Scope of Testing is about identifying the correct test cases for automation

The steps involved are:

- Identify various factors that form the basis of identifying the candidate test cases
- Apply 'Divide & rule' strategy : Break the application into smaller modules
- Analyze each module to identify the candidate test cases
- Calculate ROI

Factors influencing the scope of testing :

- In small projects

  Test case writing
  Test case execution
  Regression testing

- In Large projects

  Setting up test bed
  Generating test data, test scripts, etc.

1.4 Test Process

Testing is a process rather than a single activity.

The quality and effectiveness of software testing is primarily determined by the quality of the test processes used.

The activities of testing can be divided into the following basic steps:

- Planning and Control
- Analysis and Design
- Implementation and Execution - Evaluating and Reporting
- Test Completion/Closure activities

1.4.1 Test Process in Context

The proper, specific software test process in any given situation depends on many factors.

Few of the Contextual factors that influence the test process are :

- SDLC model and project methodologies being used
- Test levels and test types being considered
- Product and project risks
- Business domain
- Operational constraints that include :
  - Budgets and resources
  - Timescales
  - Complexity
  - Contractual and regulatory requirements
- Organizational policies and practices
- Required internal and external standards

1.4.1 Test Process in Context (Cont.)

The following sections describe general aspects of organizational test processes in terms of the following:

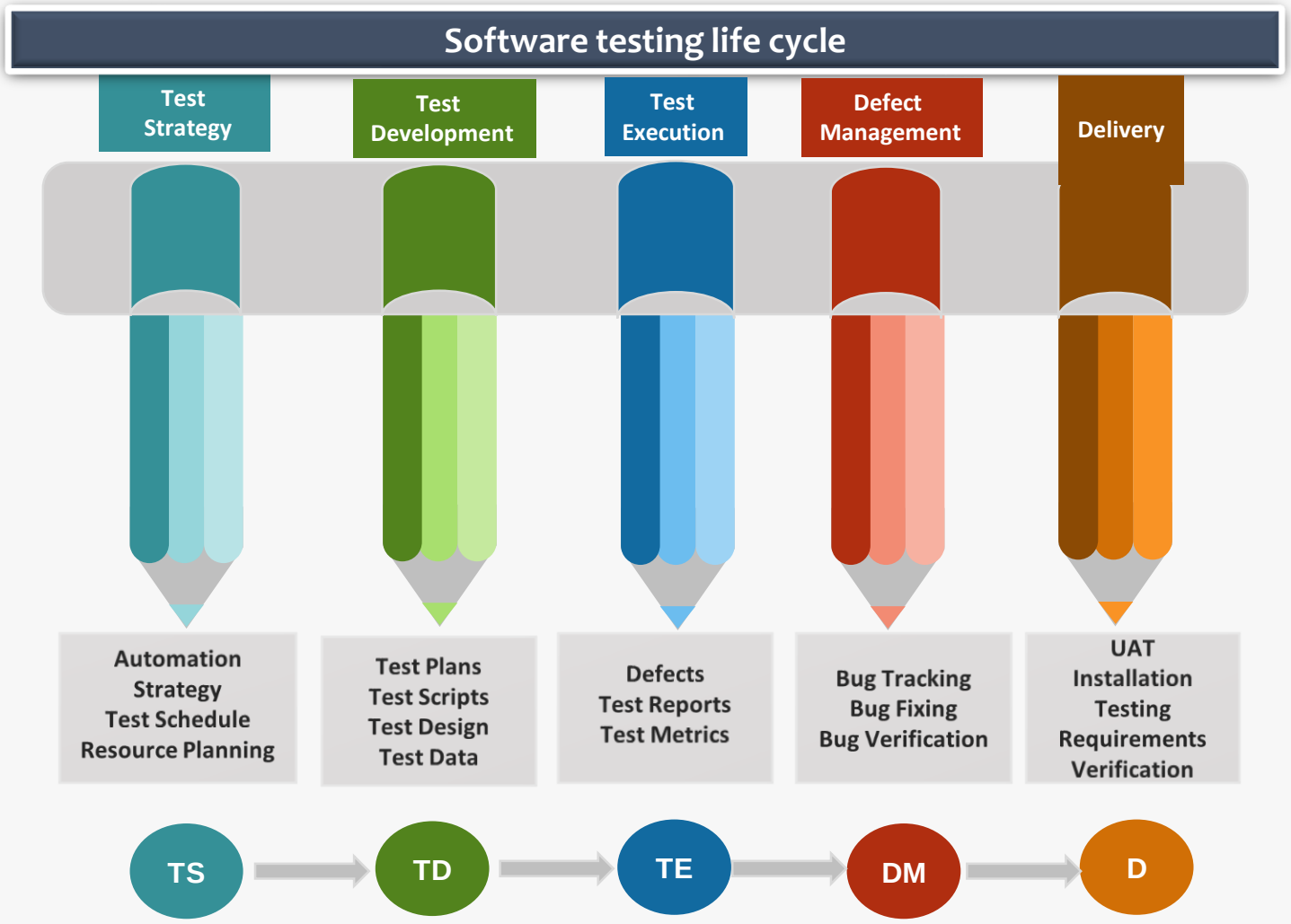
- Test activities and tasks
- Test work products
- Traceability between the test basis and test work products

It is very useful if the test basis (for any level or type of testing that is being considered) has measurable coverage criteria defined.

The coverage criteria can act effectively as key performance indicators (KPIs) to drive the activities that demonstrate achievement of software test objectives (defined in section 1.1.1).

1.4.2 Test Activities and Tasks

- Test planning
- Test monitoring and control
- Test analysis
- Test design
- Test implementation
- Test execution
- Test completion


Software testing life cycle
Test Strategy
Test Development
Test Execution
Defect Management
Delivery
Automation Strategy Test Schedule Resource Planning
Test Plans Test Scripts Test Design Test Data
Defects Test Reports Test Metrics
Bug Tracking Bug Fixing Bug Verification
UAT Installation Testing Requirements Verification
TS
TD
TE
DM
D

Test Planning

- Test planning involves activities that define the testing objectives (refer section 1.1.1) and the approach for meeting test objectives within constraints imposed by the context.

  For example, specifying suitable test techniques and tasks, and formulating a test schedule for meeting a deadline.

- Test plans may be revisited based on feedback obtained from monitoring and control activities.

Test Monitoring and Control

- Test monitoring involves an on-going comparison of actual progress against the test plan using any test monitoring metrics defined in the test plan.
- Test control involves taking actions necessary to meet the objectives of the test plan.
- Test monitoring and control are supported by the evaluation of exit criteria.
- For example, the evaluation of exit criteria for test execution as part of a given test level may include:
  - Checking test results and logs against specified coverage criteria
  - Assessing the level of component or system quality based on test results and logs
  - Determining if more tests are needed (e.g., if tests originally intended to achieve a certain level of product risk coverage failed to do so, requiring additional tests to be written and executed)

Test Analysis

Test analysis determines “what to test” in terms of measurable coverage criteria.

Test analysis includes the following major activities:

1. Analyzing the test basis appropriate to the test level being considered.
   - Requirement specifications, such as business requirements, functional requirements, system requirements, user stories, epics, use cases, or similar work products that specify desired functional and non-functional component or system behavior
   - Design and implementation information, such as system or software architecture diagrams or documents, design specifications, call flows, modelling diagrams (e.g., UML or ER diagrams), interface specifications, or similar work products that specify component or system structure
   - The implementation of the component or system itself, including code, database metadata and queries, and interfaces
   - Risk analysis reports, which may consider functional, non-functional, and structural aspects of the component or system

Test Analysis (Cont.)

2. Evaluating the test basis and test items to identify defects of various types, such as:
   - Ambiguities
   - Omissions
   - Inconsistencies
   - Inaccuracies
   - Contradictions
   - Superfluous statements

Test Analysis (Cont.)

3. Identifying features and sets of features to be tested
4. Defining and prioritizing test conditions for each feature based on analysis of the test basis, and considering functional, non-functional, and structural characteristics, other business and technical factors, and levels of risks
5. Capturing bi-directional traceability between each element of the test basis and the associated test conditions (see sections 1.4.3 and 1.4.4)

Test Design

- During test design, the test conditions are elaborated into high-level test cases, sets of high-level test cases, and other testware. So, test analysis answers the question “what to test?” while test design answers the question “how to test?”
- Test design includes the following major activities:
- Designing and prioritizing test cases and sets of test cases
- Identifying necessary test data to support test conditions and test cases
- Designing the test environment and identifying any required infrastructure and tools
- Capturing bi-directional traceability between the test basis, test conditions, test cases, and test procedures (see section 1.4.4)

The elaboration of test conditions into test cases and sets of test cases during test design often involves using test techniques (see chapter 4).

Test Implementation

- During test implementation, the testware necessary for test execution is created and/or completed, including sequencing the test cases into test procedures.
- Test design answers the question “how to test?” while test implementation answers the question “do we now have everything in place to run the tests?”

Test Implementation (Cont.)

Test implementation includes the following major activities:

- Developing and prioritizing test procedures, and, potentially, creating automated test scripts
- Creating test suites from the test procedures and automated test scripts
- Arranging the test suites within a test execution schedule in a way that results in efficient test execution (see section 5.2.4)
- Building the test environment (including, potentially, test harnesses, service virtualization, simulators, and other infrastructure items) and verifying that everything needed has been set up correctly
- Preparing test data and ensuring it is properly loaded in the test environment
- Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test suites (see section 1.4.4)

Test Execution

During test execution, test suites are run in accordance with the test execution schedule.

Test execution includes the following major activities:

- Recording the IDs and versions of the test item(s) or test object, test tool(s), and testware
- Executing tests either manually or by using test execution tools
- Comparing actual results with expected results
- Analyzing anomalies to establish their likely causes (e.g., failures may occur due to defects in the code, but false positives also may occur (see section 1.2.3))
- Reporting defects based on the failures observed (see section 5.6)
- Logging the outcome of test execution (e.g., pass, fail, blocked)
- Repeating test activities either as a result of action taken for an anomaly, or as part of the planned testing (e.g., confirmation testing, or regression testing)
- Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test results.

Test Completion

Test completion includes the following major activities:

- Checking whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution
- Creating a test summary report to be communicated to stakeholders
- Finalizing and archiving the test environment, the test data, the test infrastructure, and other testware for later reuse
- Handing over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use
- Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects
- Using the information gathered to improve test process maturity.

1.4.3 Test Work Products

Test work products are created as part of the test process. Many of the test work products can be captured and managed using test management tools and defect management tools (see chapter 6).

- Test planning work products
- Test monitoring and control work products
- Test analysis work products
- Test design work products
- Test implementation work products
- Test execution work products
- Test completion work products

1.4.4 Traceability between Test Basis and Test Work Products

It is important to establish and maintain traceability throughout the test process between each element of the test basis and the various test work products associated with it.

In addition to the evaluation of test coverage, good traceability supports:

- Analyzing the impact of changes
- Making testing auditable
- Meeting IT governance criteria
- Improving the understandability of test progress reports and test summary reports to include the status of elements of the test basis (e.g., requirements that passed their tests, requirements that failed their tests, and requirements that have pending tests)
- Relating the technical aspects of testing to stakeholders in terms that they can understand
- Providing information to assess product quality, process capability, and project progress against business goals.

Attributes of a good Tester

- A good test engineer has a 'test to break' attitude
- Need to take a different view, a different mindset (“What if it isn’t?”, “What could go wrong?”)
- An ability to understand the point of view of the customer
- A passion for quality and attention to detail
- Notice little things that others miss/ignore (See symptom not bug)
- Ability to communicate fault information to both technical (developers) and non-technical (managers and customers)
- Tact and diplomacy for maintaining a cooperative relationship with developers
- Work under worst time pressure (at the end)
- “Patience”

1.5 Psychology of Testing

- Identifying defects during a static test or identifying failures during dynamic test execution, was perceived as criticism of the product and of its author.
- Since developers expect their code to be correct, they have a confirmation bias that makes it difficult to accept that the code is incorrect.
- As a result of these psychological factors, some people may perceive testing as a destructive activity, even though it contributes greatly to project progress and product quality
- To try to reduce these perceptions and tensions between the testers and the analysts, product owners, designers, and developers, the information about defects and failures should be communicated in a constructive way.

1.5.1 Human Psychology and Testing

Testers and test managers need to have good interpersonal skills to be able to communicate effectively about defects, failures, test results, test progress, and risks, and to build positive relationships with colleagues.

Ways to communicate well include the following examples:

- Start with collaboration rather than battles. Remind everyone of the common goal of better quality systems.
- Emphasize on the benefits of testing. For example, for the authors, defect information can help them improve their work products and their skills. For the organization, defects found and fixed during testing will save time and money and reduce overall risk to product quality.
- Communicate test results and other findings in a neutral, fact-focused way without criticizing the author.
- Confirm that the other person has understood what has been said and vice versa.
- Clearly defining the right set of test objectives.

Code of Ethics for Tester

Involvement in software testing enables individuals to learn confidential and privileged information. A code of ethics is therefore necessary, among other reasons to ensure that the information is not put to inappropriate use.

PUBLIC - Tester shall act consistently with public interest

CLIENT AND EMPLOYER - Tester shall act in best interests of their client and employer

PRODUCT - Tester shall ensure that the deliverables they provide meet highest professional standards possible

JUDGMENT - Tester shall maintain integrity and independence in their professional judgment

MANAGEMENT - Tester managers and leaders shall subscribe to and promote an ethical approach to manage software testing

PROFESSION - Tester shall advance the integrity and reputation of the profession consistent with the public interest

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

SELF - Tester shall participate in lifelong learning and shall promote an ethical approach to the practice of the profession

1.5.2 Tester's and Developer's Mindsets

- Developers and testers often think differently. Bringing these mindsets together helps to achieve a higher level of product quality.
- A mindset reflects an individual's assumptions and preferred methods for decision making and problem solving.
- A tester's mindset should include curiosity, professional pessimism, a critical eye, attention to detail, and a motivation for good and positive communications and relationships.
- A tester's mindset tends to grow and mature as the tester gains experience.
- A developer's mindset may include designing and building solutions than in contemplating what might be wrong with those solutions.
- In addition, confirmation bias makes it difficult to find mistakes in their own work. With the right mindset, developers are able to test their own code.
- Having some of the test activities done by independent testers increases defect detection effectiveness, which is particularly important for large, complex, or safety-critical systems.

Limitations of Software Testing

Even if we could generate the input, run the tests, and evaluate the output, we would not detect all faults

Correctness is not checked

- The programmer may have misinterpreted the specs, the specs may have misinterpreted the requirements

There is no way to find missing paths due to coding errors

Summary

In this lesson, you have learnt:

- Testing is an extremely creative & intellectually challenging task
- No software exists without bug
- Testing is conducted with the help of users requirements, design documents, functionality, internal structures & design, by executing code
- Scope of testing
- The cost of not testing is potentially much higher
- Testing is in a way a destructive process
- A successful test case is one that brings out an error in program
- Various principles of testing

Review Question

Question 1: What is visible to end-users is a deviation from the specific or expected behavior is called as

- Defect
- Bug
- Failure
- Fault

Question 2: _____ is a planned sequence of actions.

Question 3: Pick the best definition of Quality :

- Quality is job done
- Zero defects
- Conformance to requirements
- Work as designed

Question 4: One cannot test a program completely to guarantee that it is error free (T/F)

Question 5: One can find missing paths due to coding errors (T/F)

Testing Concepts

Lesson 2: Testing throughout the Software Development Life Cycle

Lesson Objectives

To understand the following topics :

- Software Development Lifecycle Models
  - Software Development and Software Testing
  - Software Development Lifecycle Models in Context
- Test Levels
  - Component Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing

Lesson Objectives

- Test Types
  - Functional Testing
  - Non-functional Testing
  - White-box Testing
  - Change-related Testing
  - Test Types and Test Levels
- Maintenance Testing
  - Triggers for Maintenance
  - Impact Analysis for Maintenance
- Test Case Terminologies
- Test Data

2.1 Software Development Life Cycle (SDLC) Models

Testing is not a stand-alone activity

It has its place within a SDLC model

In any SDLC model, a part of testing is focused on Verification and a part is focused on Validation

- Verification: Is the deliverable built according to the specification?
- Validation: Is the deliverable fit for purpose?

2.1.1 Software Development & Software Testing

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

- For every development activity, there is a corresponding test activity
- Each test level has test objectives specific to that level
- Test analysis and design for a given test level begin during the corresponding development activity
- Testers participate in discussions to define and refine requirements and design, and are involved in reviewing work products (e.g., requirements, design, user stories, etc.) as soon as drafts are available

No matter which SDLC model is chosen, test activities should start in the early stages of the lifecycle, adhering to the testing principle of early testing.

2.1.2 Software Development Lifecycle Models in Context

- SDLC models must be selected and adapted based on the context of project and product characteristics such as - project goal, the type of product being developed, business priorities (e.g., time-tomarket), and identified product and project risks.

Example : The development and testing of a minor internal administrative system should differ from the development and testing of a safety-critical system such as an automobile's brake control system.

Example : In some cases organizational and cultural issues may inhibit communication between team members, which can impede iterative development.

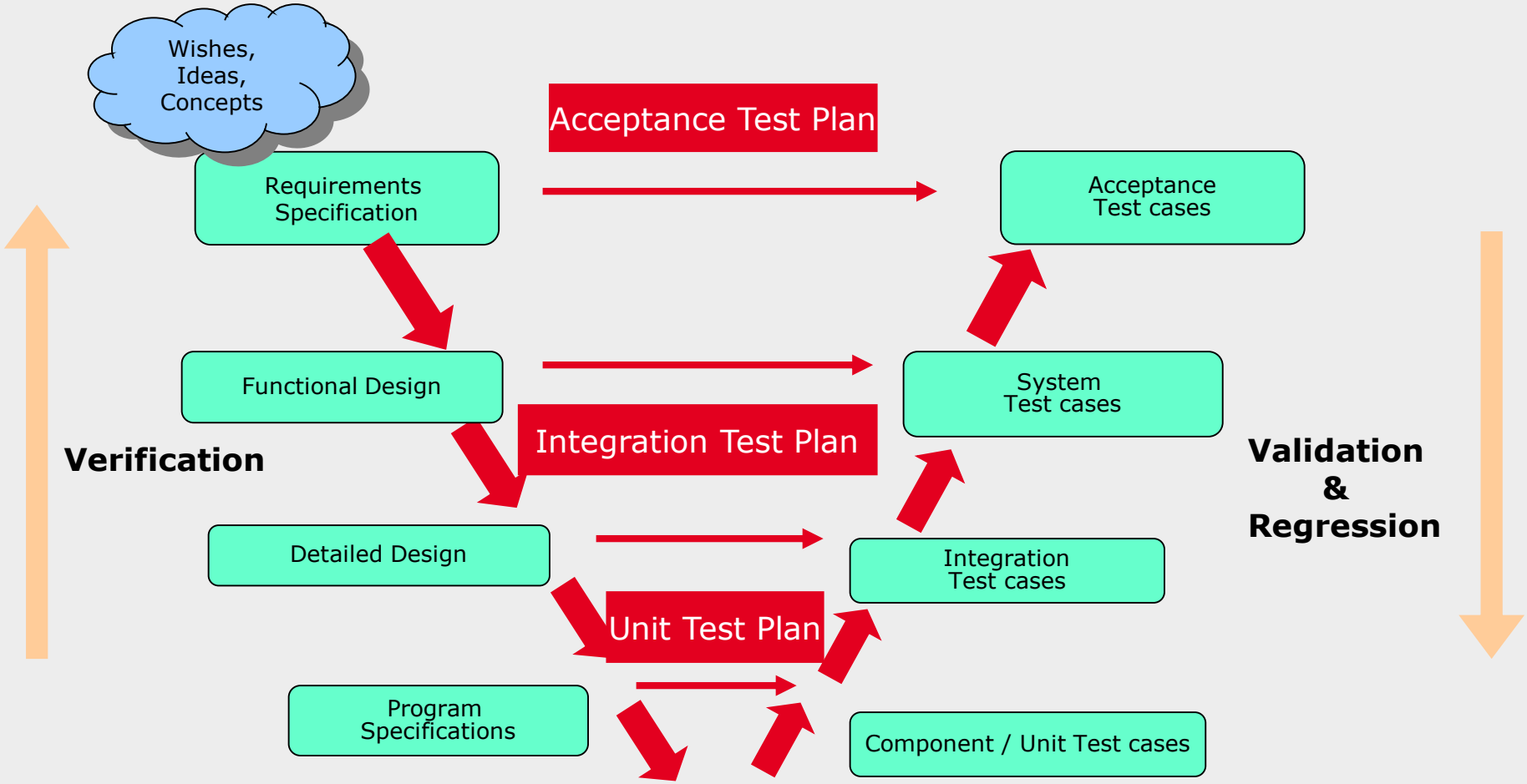
2.1.2 Software Development Lifecycle Models in Context (Cont.)

- Depending on the context of the project, it may be necessary to combine or reorganize test levels and/or test activities.

  Example : For the integration of a commercial off-the-shelf (COTS) software product into a larger system, the purchaser may perform interoperability testing at the system integration test level (e.g., integration to the infrastructure and other systems) and at the acceptance test level (functional and non-functional, along with user acceptance testing and operational acceptance testing).

2.2 Test Levels in V-Model

- Test levels are groups of test activities that are organized and managed together. Each test level is an instance of the test process related to other activities with in SDLC.



Verification and Validation

Verification

- Verification refers to a set of activities which ensures that software correctly implements a specific function.
- Purpose of verification is to check: Are we building the product right?
- Example: code and document reviews, inspections, walkthroughs.
- It is a Quality improvement process.
- It is involve with the reviewing and evaluating the process.
- It is conducted by QA team.
- Verification is Correctness.

Verification and Validation (Cont.)

Validation

- Validation is the following process of verification.
- Purpose of Validation is to check : Are we building the right product?
- Validation refers to a different set of activities which ensures that the software that has been built is traceable to customer requirements.
- After each validation test has been conducted, one of two possible conditions exist:
  1. The function or performance characteristics conform to specification and are accepted, or
  2. Deviation from specification and a deficiency list is created.
- It is conducted by development team with the help from QC team.

2.2 Test Levels (Cont.)

Unit (Component) testing

- Unit testing is code-based and performed primarily by developers to demonstrate that their smallest pieces of executable code function suitably.

Integration testing

- Integration testing demonstrates that two or more units or other integrations work together properly, and tends to focus on the interfaces specified in low-level design.

System testing

- System testing demonstrates that the system works end-to-end in a production-like environment to provide the business functions specified in the high-level design.

Acceptance testing

- Acceptance testing is conducted by business owners and users to confirm that the system does, in fact, meet their business requirements.

2.2 Test Levels (Cont.)

Test levels are characterized by the following attributes:

- Specific objectives
- Test basis, referenced to derive test cases
- Test object (i.e., what is being tested)
- Typical defects and failures
- Specific approaches and responsibilities.

For every test level, a suitable test environment is required.

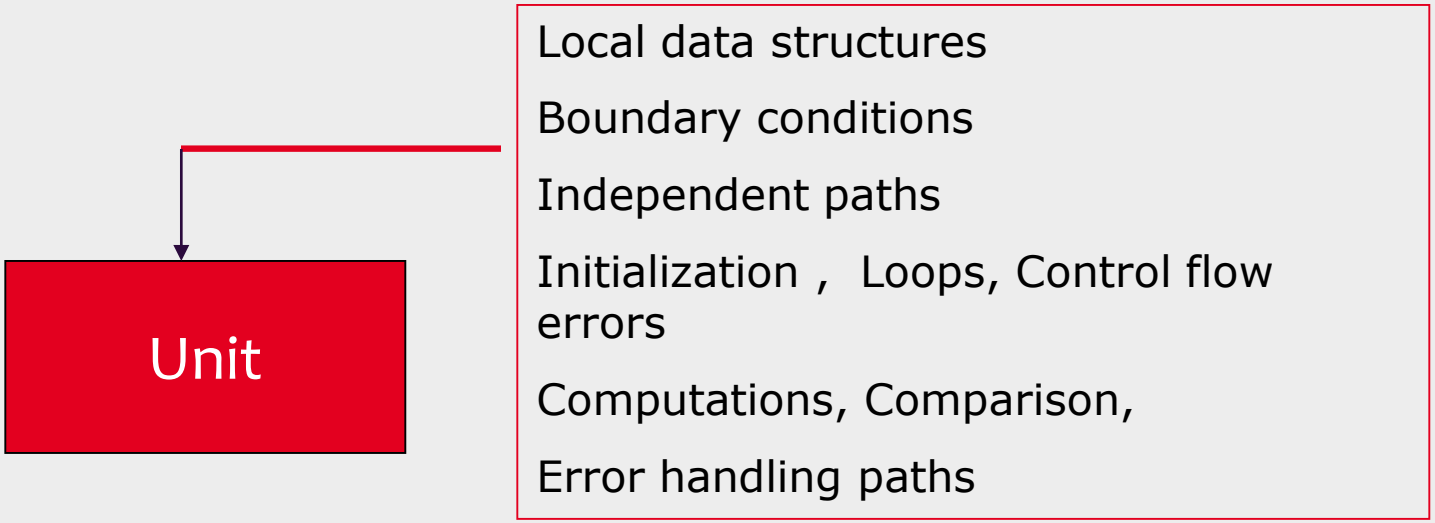
Example : In acceptance testing, for example, a production-like test environment is ideal, while in component testing the developers typically use their own development environment.

2.2.1 Component Testing

- The most 'micro' scale of testing to test particular functions, procedures or code modules or components is called Component testing ; Also called as Module or Unit testing
- Typically done by the programmer and not by Test Engineers, as it requires detailed knowledge of the internal program design and code.
- Objectives of component testing include:
  - Reducing risk
  - Verifying whether the functional and non-functional behaviors of the component are as designed and specified
  - Building confidence in the component’s quality
  - Finding defects in the component
  - Preventing defects from escaping to higher test levels

Component /Unit Testing

Unit testing uncovers errors in logic and function within the boundaries of a component.



Component Testing

Test Basis : Test basis for component testing include:

- Detailed design
- Code
- Data model
- Component specifications

Test objects : Typical test objects for component testing include:

- Components, units or modules
- Code and data structures
- Classes
- Database modules

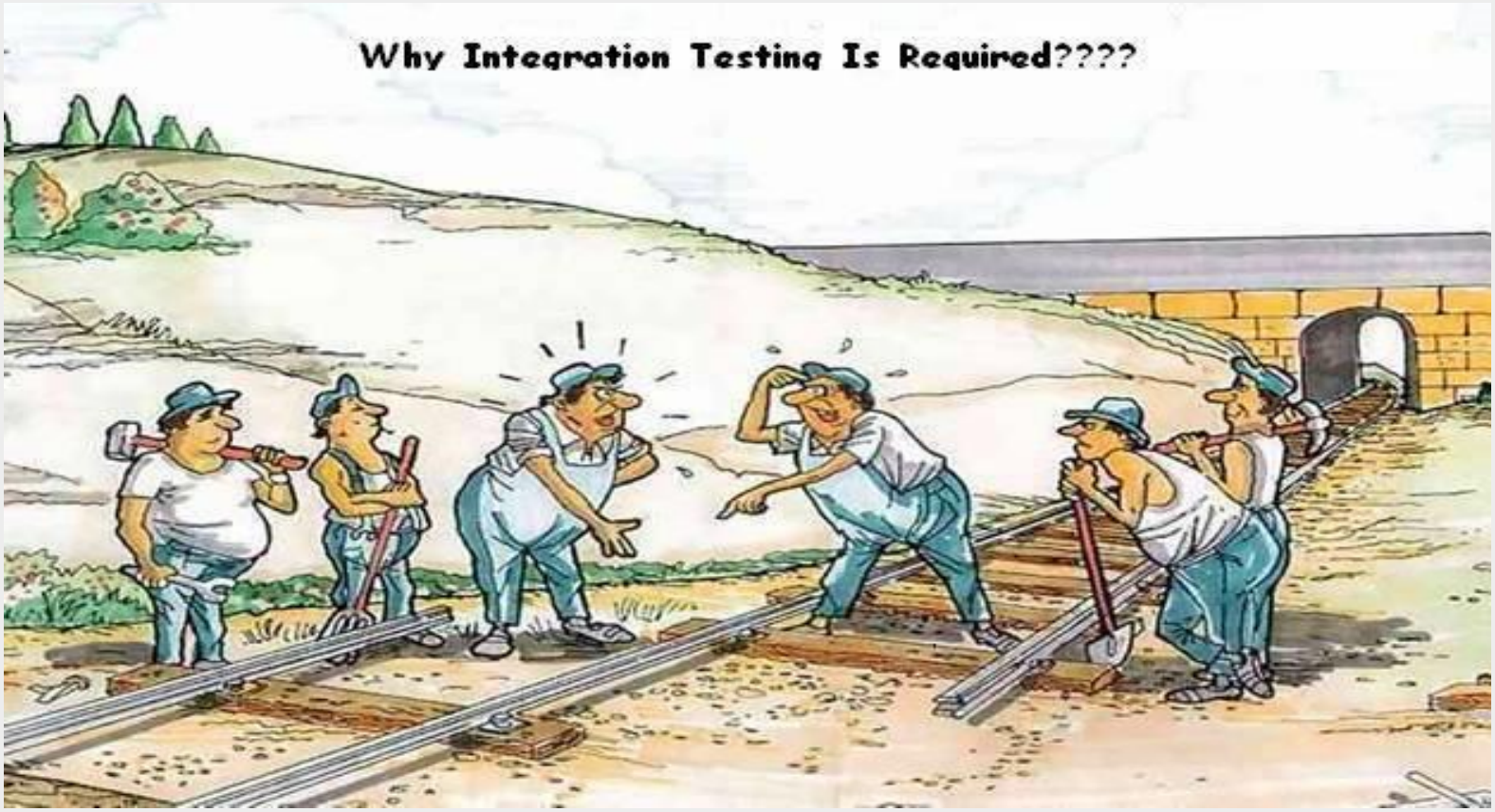
Typical defects and failures : Typical defects and failures for component testing include :

- Incorrect functionality (e.g., not as described in design specifications)
- Data flow problems
- Incorrect code and logic

2.2.2 Integration testing

- Integration testing focuses on interactions between components or systems.
- Objectives of Integration testing include:
  - Reducing risk
  - Verifying whether the functional and non-functional behaviors of the interfaces are as designed and specified
  - Building confidence in the quality of the interfaces.
  - Finding defects (which may be in the interfaces themselves or within the components or systems)
  - Preventing defects from escaping to higher test levels

Why Integration Testing is Required?



Two Levels of Integration testing

There are two different levels of integration testing which may be carried out on test objects of varying size as follows:

- **Component integration testing (CIT)** focuses on the interactions and interfaces between integrated components. Component integration testing is performed after component testing.

- **System integration testing (SIT)** focuses on the interactions and interfaces between systems, packages, and micro services. System integration testing may be done after system testing.

Integration Testing

Test Basis : Test basis for Integration testing include:

- Software and system design
- Sequence diagrams
- Interface and communication protocol specifications
- Use cases
- Architecture at component or system level
- Workflows
- External interface definitions

Test objects : Typical test objects for Integration testing include:

- Subsystems
- Databases
- Infrastructure
- Interfaces
- APIs
- Micro services

Integration Testing

Typical defects and failures for CIT include :

- Incorrect data, missing data, or incorrect data encoding
- Incorrect sequencing or timing of interface calls
- Interface mismatch
- Failures in communication between components
- Unhandled or improperly handled communication failures between components
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between components

Integration Testing

Typical defects and failures for SIT include:

- Inconsistent message structures between systems
- Incorrect data, missing data, or incorrect data encoding
- Interface mismatch
- Failures in communication between systems
- Unhandled or improperly handled communication failures between systems
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between systems
- Failure to comply with mandatory security regulations

Types of Integration testing

Modules are integrated by two ways.

1. **Non-incremental Testing (Big Bang Testing)**
   - Each Module is tested independently and at the end, all modules are combined to form a application
1. **Incremental Module Testing.**
   - There are two types by which incremental module testing is achieved.
     - **Top down Approach :** Firstly top module is tested first. Once testing of top module is done then any one of the next level modules is added and tested. This continues till last module at lowest level is tested and it is called as Stub.
     - **Bottom up Approach :** Firstly module at the lowest level is tested first. Once testing of that module is done then any one of the next level modules is added to it and tested. This continues till top most module is added to rest all and tested and it is called as Driver.

2.2.3 System Testing

- System testing focuses on the behavior and capabilities of a whole system or product, often considering the end-to-end tasks the system can perform and the non-functional behaviors it exhibits while performing those tasks.
- Test the software in the real environment in which it is to operate. (hardware, people, information, etc.)
- Observe how the system performs in its target environment, for example in terms of speed, with volumes of data, many users, all making multiple requests.
- Test how secure the system is and how can the system recover if some fault is encountered in the middle of procession.
- System Testing, by definition, is impossible if the project has not produced a written set of measurable objectives for its product.

System testing

- Objectives of System testing include:
  - Reducing risk
  - Verifying whether the functional and non-functional behaviors of the system are as designed and specified
  - Validating that the system is complete and will work as expected
  - Building confidence in the quality of the system as a whole
  - Finding defects
  - Preventing defects from escaping to higher test levels or production

System Testing

Test Basis : Test basis for System testing include:

- System and software requirement specifications (functional and non-functional)
- Risk analysis reports
- Use cases
- Epics and user stories
- Models of system behavior
- State diagrams
- System and user manuals

Test objects : Typical test objects for System testing include:

- Applications
- Hardware/software systems
- Operating systems
- System under test (SUT)
- System configuration and configuration data

System Testing

Typical defects and failures :

- Typical defects and failures for System Testing include :
  - Incorrect calculations
  - Incorrect or unexpected system functional or non-functional behavior
  - Incorrect control and/or data flows within the system
  - Failure to properly and completely carry out end-to-end functional tasks
  - Failure of the system to work properly in the production environment(s)
  - Failure of the system to work as described in system and user manuals

Types of System Testing

- Functional Testing
- Performance Testing
- Volume Testing
- Load Testing
- Stress Testing
- Security Testing
- Web Security Testing
- Localization Testing
- Usability Testing
- Recovery Testing
- Documentation Testing
- Configuration Testing
- Installation Testing
- User Acceptance Testing
- Testing related to Changes : Re-Testing and Regression Testing
- Re-testing (Confirmation Testing)
- Regression Testing
- Exploratory Testing
- Maintenance Testing

Functional Testing

The main objective of functional testing is to verify that each function of the software application / system operates in accordance with the written requirement specifications.

It is a black-box process :

- Is not concerned about the actual code
- Focus is on validating features
- Uses external interfaces, including Application programming interfaces (APIs), Graphical user interfaces (GUIs) and Command line interfaces (CLIs)

Testing functionality can be done from two perspectives :

1. Business-process-based testing uses knowledge of the business processes
2. Requirements-based testing uses a specification of the functional requirements for the system as the basis for designing tests

Performance Testing

Performance

- Performance is the behavior of the system w.r.t. goals for time, space, cost and reliability

Performance objectives:

- **Throughput :** The number of tasks completed per unit time. Indicates how much work has been done within an interval
- **Response time :** The time elapsed during input arrival and output delivery
- **Utilization :** The percentage of time a component (CPU, Channel, storage, file server) is busy

Volume Testing

This testing is subjecting the program to heavy volumes of data. For e.g.

- A compiler would be fed a large source program to compile
- An operating systems job queue would be filled to full capacity
- A file system would be fed with enough data to cause the program to switch from one volume to another.

Load Testing

Volume testing creates a real-life end user pressure for the target software. This tests how the software acts when numerous end users access it concurrently. For e.g.

- Downloading a sequence of huge files from the web
- Giving lots of work to a printer in a line

Stress Testing

- Stress testing involves subjecting the program to heavy loads or stresses.
- The idea is to try to “break” the system.
- That is, we want to see what happens when the system is pushed beyond design limits.
- It is not same as volume testing.
- A heavy stress is a peak volume of data encounters over a short time.
- In Stress testing a considerable load is generated as quickly as possible in order to stress the application and analyze the maximum limit of concurrent users the application can support.

Stress Testing(Cont.)

Stress tests executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

Example :

- Generate 5 interrupts when the average rate is 2 or 3
- Increase input data rate
- Test cases that require max. memory

Stress Tests should answer the following questions

- Does the system degrade gently or does the server shut down
- Are appropriate messages displayed ? E.g. Server not available
- Are transactions lost as capacity is exceeded
- Are certain functions discontinued as capacity reaches the 80 or 90 percent level

Security Testing

Security Testing verifies that protection mechanisms built into the system will protect it from improper penetration.

Security testing is the process of executing test cases that subvert the program's security checks.

Example :

- One tries to break the operating systems memory protection mechanisms
- One tries to subvert the DBMS's data security mechanisms
- The role of the developer is to make penetration cost more than the value of the information that will be obtained

Web Security Testing

Web application security is a branch of Information Security that deals specifically with security of web applications.

It provides a strategic approach in identifying, analyzing and building a secure web applications.

It is performed by Web Application Security Assessment.

Localization Testing

Localization translates the product UI and occasionally changes some settings to make it suitable for another region.

The test effort during localization testing focuses on

- Areas affected during localization, UI and content
- Culture/locale-specific, language specific and region specific areas

Usability Testing

Usability is

- The effectiveness, efficiency and satisfaction with which specified users can achieve specified goals in a particular environment ISO 9241-11
- Effective–- Accomplishes user’s goal
- Efficient-- Accomplishes the goal quickly
- Satisfaction–- User enjoys the experience

Test Categories and objectives

- Interactivity (Pull down menus, buttons)
- Layout
- Readability
- Aesthetics
- Display characteristics
- Time sensitivity
- Personalization

Usability Testing (Cont.)

Using specialized Test Labs a rigorous testing process is conducted to get quantitative and qualitative data on the effectiveness of user interfaces

Representative or actual users are asked to perform several key tasks under close observation, both by live observers and through video recording

During and at the end of the session, users evaluate the product based on their experiences

Recovery Testing

A system test that forces the software to fail in variety of ways, checks performed

- recovery is automatic (performed by the system itself)
- reinitialization
- check pointing mechanisms
- data recovery
- restarts are evaluated for correctness

This test confirms that the program recovers from expected or unexpected events. Events can include shortage of disk space, unexpected loss of communication

Documentation Testing

This testing is done to ensure the validity and usability of the documentation

This includes user Manuals, Help Screens, Installation and Release Notes

Purpose is to find out whether documentation matches the product and vice versa

Well-tested manual helps to train users and support staff faster

Configuration Testing

Attempts to uncover errors that are specific to a particular client or server environment.

Create a cross reference matrix defining all probable operating systems, browsers, hardware platforms and communication protocols.

Test to uncover errors associated with each possible configuration

Installation Testing

Installer is the first contact a user has with a new software!!!

Installation testing is required to ensure:

- Application is getting installed properly
- New program that is installed is working as desired
- Old programs are not hampered
- System stability is maintained
- System integrity is not compromised

2.2.4 Acceptance testing

- Acceptance testing, like system testing, typically focuses on the behavior and capabilities of a whole system or product
- Objectives of Acceptance testing include:
  - Establishing confidence in the quality of the system as a whole
  - Validating that the system is complete and will work as expected
  - Verifying that functional and non-functional behaviors of the system are as specified

Forms of Acceptance testing

Common forms of acceptance testing include the following:

- User Acceptance testing (UAT)
- Operational Acceptance testing (OAT)
- Contractual and Regulatory Acceptance testing
- Alpha and Beta testing.

User Acceptance Testing (UAT)

- A test executed by the end user(s) is typically focused on validating the fitness for use of the system by intended users in a real or simulated operational environment. The main objective is building confidence that the users can use the system to meet their needs, fulfill requirements, and perform business processes with minimum difficulty, cost, and risk.

- Usually carried out by the end user to test whether or not the right system has been created

Operational Acceptance Testing (OAT)

- Acceptance testing of the system by operations or systems administration staff is usually performed in a (simulated) production environment.
- The tests focus on operational aspects, and may include:
  - Testing of backup and restore
  - Installing, uninstalling and upgrading
  - Disaster recovery
  - User management
  - Maintenance tasks
  - Data load and migration tasks
  - Checks for security vulnerabilities
  - Performance testing

Contractual and Regulatory Testing

- **Contractual** acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance criteria should be defined when the parties agree to the contract. Contractual acceptance testing is often performed by users or by independent testers.

- **Regulatory** acceptance testing is performed against any regulations that must be adhered to, such as government, legal, or safety regulations. Regulatory acceptance testing is often performed by users or by independent testers, sometimes with the results being witnessed or audited by regulatory agencies.

Alpha and Beta Testing

- Alpha and beta testing are typically used by developers of commercial off-the-shelf (COTS) software who want to get feedback from potential or existing users, customers, and/or operators before the software product is put on the market.
- Alpha testing is performed at the developing organization's site, not by the development team, but by potential or existing customers, and/or operators or an independent test team.
- Beta testing is performed by potential or existing customers, and/or operators at their own locations. Beta testing may come after alpha testing, or may occur without any preceding alpha testing having occurred.

Alpha and Beta Testing

Test Basis

Examples of work products that can be used as a test basis for any form of acceptance testing include:

- Business processes
- User or business requirements
- Regulations, legal contracts and standards
- Use cases
- System requirements
- System or user documentation
- Installation procedures
- Risk analysis reports

Alpha and Beta Testing

Test Objects

Typical test objects for any form of acceptance testing include:

- System under test
- System configuration and configuration data
- Business processes for a fully integrated system
- Recovery systems and hot sites (for business continuity and disaster recovery testing)
- Operational and maintenance processes
- Forms
- Reports
- Existing and converted production data

Alpha and Beta Testing

Typical Defects and Failures

Typical defects for any form of acceptance testing include:

- System workflows do not meet business or user requirements
- Business rules are not implemented correctly
- System does not satisfy contractual or regulatory requirements
- Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform

2.3 Test Types

A test type is a group of test activities aimed at testing specific characteristics of a software system based on specific test objectives such as :

- Evaluating functional quality characteristics, such as completeness, correctness, and appropriateness
- Evaluating non-functional quality characteristics, such as reliability, performance efficiency, security, compatibility, and usability
- Evaluating whether the structure or architecture of the component or system is correct, complete, and as specified
- Evaluating the effects of changes, such as confirming that defects have been fixed (confirmation testing) and looking for unintended changes in behavior resulting from software or environment changes (regression testing)

2.3.1 Functional Testing

- Functional testing of a system involves tests that evaluate functional requirements such as business requirements specifications, epics, user stories, use cases, or functional specifications, or they may be undocumented.
- The functions are “what” the system should do
- Functional tests should be performed at all test levels though the focus is different at each level.
- Black-box techniques may be used to derive test conditions and test cases for the functionality of the component or system

2.3.2 Non-Functional Testing

- Non-functional testing of a system evaluates characteristics of systems and software such as usability, performance efficiency or security.
- Non-functional testing is the testing of “how well” the system behaves.
- non-functional testing can and often should be performed at all test levels, and done as early as possible.
- Black-box techniques may be used to derive test conditions and test cases even for nonfunctional testing too. For example, boundary value analysis can be used to define the stress conditions for performance tests.

2.3.3 White-box Testing

- White-box testing derives tests based on the system’s internal structure or implementation. Internal structure may include code, architecture, work flows, and/or data flows within the system.

- White-box testing derives tests based on the system’s internal structure or implementation. Internal structure may include code, architecture, work flows, and/or data flows within the system.

2.3.4 Change-related Testing

- When changes are made to a system, either to correct a defect or because of new or changing functionality, testing should be done to confirm that the changes have corrected the defect or implemented the functionality correctly, and have not caused any unforeseen adverse consequences.
  - Confirmation testing
  - Regression testing
- Confirmation testing and regression testing are performed at all test levels.
  - Especially in iterative and incremental development lifecycles (e.g., Agile), new features, changes to existing features, and code refactoring results in frequent changes to the code, which requires change-related testing.
  - Due to the evolving nature of the system (e.g. IoT system), confirmation and regression testing are very important.

Re-testing (Confirmation Testing)

- After a defect is fixed, the software is tested by re-running all test cases that failed due to the defect, within same environment, with same inputs and same preconditions on the new software version.
- The software may also be tested with new tests if, for instance, the defect was missing functionality.
- At the very least, the steps to reproduce the failure(s) caused by the defect must be re-executed on the new software version.
- The purpose of a confirmation test is to confirm whether the original defect has been successfully fixed

Regression Testing

- It is possible that a change made in one part of the code, whether a fix or another type of change, may accidentally affect the behavior of other parts of the code, whether within the same component, in other components of the same system, or even in other systems.
- Changes may include changes to the environment, such as a new version of an operating system or database management system.
- Such unintended side-effects are called regressions.
- Regression testing involves running tests to detect such unintended side-effects

2.3.5 Test Types and Test Levels
Example: Banking Application

It is possible to perform any of the test types at any test level.

Examples of Functional Tests :

- For CT, tests are designed based on how a component should calculate compound interest.
- For CIT, tests are designed based on how account information captured at the user interface is passed to the business logic.
- For ST, tests are designed based on how account holders can apply for a line of credit on their checking accounts.
- For SIT, tests are designed based on how the system uses an external micro service to check an account holder’s credit score.
- For UAT, tests are designed based on how the banker handles approving or declining a credit application.

2.3.5 Test Types and Test Levels (Cont..)
Example: Banking Application

Examples of Non-Functional Tests :

- For CT, performance tests are designed to evaluate the number of CPU cycles required to perform a complex total interest calculation
- For CIT, security tests are designed for buffer overflow vulnerabilities due to data passed from the user interface to the business logic.
- For ST, portability tests are designed to check whether the presentation layer works on all supported browsers and mobile devices.
- For SIT, reliability tests are designed to evaluate system robustness if the credit score micro service fails to respond.
- For UAT, usability tests are designed to evaluate the accessibility of the banker’s credit processing interface for people with disabilities.

2.3.5 Test Types and Test Levels (Cont..)
Example: Banking Application

Examples of White-box Tests :

- For CT, performance tests are designed to achieve complete statement and decision coverage for all components that perform financial calculations.
- For CIT, security tests are designed to exercise how each screen in the browser interface passes data to the next screen and to the business logic.
- For ST, tests are designed to cover sequences of web pages that can occur during a credit line application.
- For SIT, tests are designed to exercise all possible inquiry types sent to the credit score microservice.
- For UAT, tests are designed to cover all supported financial data file structures and value ranges for bank-to-bank transfers.

2.3.5 Test Types and Test Levels (Cont..)
Example: Banking Application

Examples of change-related Tests :

- For CT, automated regression tests are built for each component and included within the continuous integration framework.
- For CIT, tests are designed to confirm fixes to interface-related defects as the fixes are checked into the code repository.
- For ST, all tests for a given workflow are re-executed if any screen on that workflow changes.
- For SIT, tests of the application interacting with the credit scoring micro service are re-executed daily as part of continuous deployment of that micro service.
- For UAT, all previously-failed tests are re-executed after a defect found in acceptance testing is fixed.

2.4 Maintenance Testing

- Once deployed to production environments, software and systems need to be maintained.

- Changes of various sorts are almost inevitable in delivered software and systems, either to fix defects discovered in operational use, to add new functionality, or to delete or alter already-delivered functionality.

- Maintenance is also needed to preserve or improve non-functional quality characteristics of the component or system over its lifetime, especially performance efficiency, compatibility, reliability, security, compatibility, and portability.

2.4 Maintenance Testing (cont..)

- When any changes are made as part of maintenance, maintenance testing should be performed, both to evaluate the success with which the changes were made and to check for possible side-effects (e.g., regressions) in parts of the system that remain unchanged.
- Maintenance can involve planned releases and unplanned releases (hot fixes).
- Maintenance testing is required at multiple test levels, using various test types, based on its scope. The scope of maintenance testing depends on:
  - The degree of risk of the change, for example, the degree to which the changed area of software communicates with other components or systems
  - The size of the existing system
  - The size of the change

2.4.1 Triggers for Maintenance

There are several reasons (triggers) why maintenance testing takes place, both for planned and unplanned changes :

- Modification, such as planned enhancements (e.g., release-based), corrective and emergency changes, changes of the operational environment (such as planned operating system or database upgrades), upgrades of COTS software, and patches for defects and vulnerabilities
- Migration, such as from one platform to another, which can require operational tests of the new environment as well as of the changed software, or tests of data conversion when data from another application will be migrated into the system being maintained
- Retirement, such as when an application reaches the end of its life

2.4.2 Impact Analysis for Maintenance

Impact analysis may be done before a change is made, to help decide if the change should be made, based on the potential consequences in other areas of the system.

Impact analysis can be difficult if:

- Specifications (e.g., business requirements, user stories, architecture) are out of date or missing
- Test cases are not documented or are out of date
- Bi-directional traceability between tests and the test basis has not been maintained
- Tool support is weak or non-existent
- The people involved do not have domain and/or system knowledge
- Insufficient attention has been paid to the software's maintainability during development

2.5 Test Case Terminologies

Pre Condition

- Environmental and state which must be fulfilled before the component/unit can be executed with a particular input value.

Test Analysis

- is a process for deriving test information by viewing the Test Basis
- For testing, test basis is used to derive what could be tested

Test basis includes whatever the test are based on such as System Requirement

- A Technical specification
- The code itself (for structural testing)
- A business process

Test Condition

- It is a set of rules under which a tester will determine if a requirement is partially or fully satisfied
- One test condition will have multiple test cases

Test Case Terminologies (cont.)

Test Scenario

- It is an end-to-end flow of a combination of test conditions & test cases integrated in a logical sequence, covering a business processes
- This clearly states what needs to be tested
- One test condition will have multiple test cases

Test Procedure (Test Steps)

- A detailed description of steps to execute the test

Test Data/Input

- Inputs & its combinations/variables used

Expected Output

- This is the expected output for any test case or any scenario

Actual Output

- This is the actual result which occurs after executing the test case

Test Result/Status

- Pass / Fail – If the program works as given in the specification, it is said to Pass otherwise Fail.
- Failed test cases may lead to code rework

Other Terminologies

Test Suite – A set of individual test cases/scenarios that are executed as a package, in a particular sequence and to test a particular aspect

- E.g. Test Suite for a GUI or Test Suite for functionality

Test Cycle – A test cycle consists of a series of test suites which comprises a complete execution set from the initial setup to the test environment through reporting and clean up.

- E.g. Integration test cycle / regression test cycle

A good Test Case

Has a high probability of detecting error(s)
Test cases help us discover information
Maximize bug count
Help managers make ship / no-ship decisions
Minimize technical support costs
Assess conformance to specification
Verify correctness of the product
Minimize safety-related lawsuit risk
Find safe scenarios for use of the product
Assure quality

2.6 Test data

- An application is built for a business purpose. We input data and there is a corresponding output. While an application is being tested we need to use dummy data to simulate the business workflows. This is called test data.

- The test data may be any kind of input to application, any kind of file that is loaded by the application or entries read from the database tables. It may be in any format like xml test data, stand alone variables, SQL test data etc.

Properties of Good Test Data

Realistic – accurate in context of real life

- E.g. Age of a student giving graduation exam is at least 18

Practically valid – data related to business logic

- E.g. Age of a student giving graduation exam is at least 18 says that 60 years is also valid input but practically the age of a graduate student cannot be 60

Cover varied scenarios

- E.g. Don’t just consider the scenario of only regular students but also consider the irregular students, also the students who are giving a re-attempt, etc.

Exceptional data

- E.g. There may be few students who are physically handicapped must also be considered for attempting the exam

Summary

In this lesson, you have learnt:

- Verification refers to a set of activities which ensures that software correctly implements a specific function.
- Validation refers to a different set of activities which ensures that the software that has been built is traceable to customer requirements.
- Different testing phases are
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing

Review Question

Question 1: _____ is a Quality improvement process

Question 2: To test a function, the programmer has to write a _____, which calls the function to be tested and passes it test data

Question 3: Volume tests executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

- True/False

Question 4: Acceptance testing is not a responsibility of the Developing Organization

- True/False

Question 5: Difference between re-testing & regression testing is :

- re-testing is running a test again; regression testing looks for unexpected side effects
- re-testing looks for unexpected side effects; regression testing is repeating

Review Question: Match the Following

1. Beta testing	A. Volume testing
2. Response time	B. Exploratory testing
3. Aesthetics	C. Acceptance testing
	D. Documentation testing
	E. Performance testing
	F. Usability testing

Review Question: Match the Following

1. Economics of limiting	A. Driving
2. Testing	B. Exhaustive testing
3. A good test case	C. Limiting
4. Use every possible input condition as a test case	D. Test cycle
	E. Comparing outputs with specified or intended
	F. Maximize bug count

Testing Concepts

Lesson 3: Static Testing

Lesson Objectives

To understand the following topics:

- Types of Testing Techniques
- Differences between Static & Dynamic Testing
- Static Testing Basics
  - Work Products Examined by Static Testing
  - Benefits of Static Testing
- Review Process
  - Work Product Review Process
  - Roles and responsibilities in a formal review
  - Review Types
  - Applying Review Techniques – checklist based Testing in detail.
  - Success Factors for Reviews

Types of Testing Techniques

There are two types of Testing Techniques

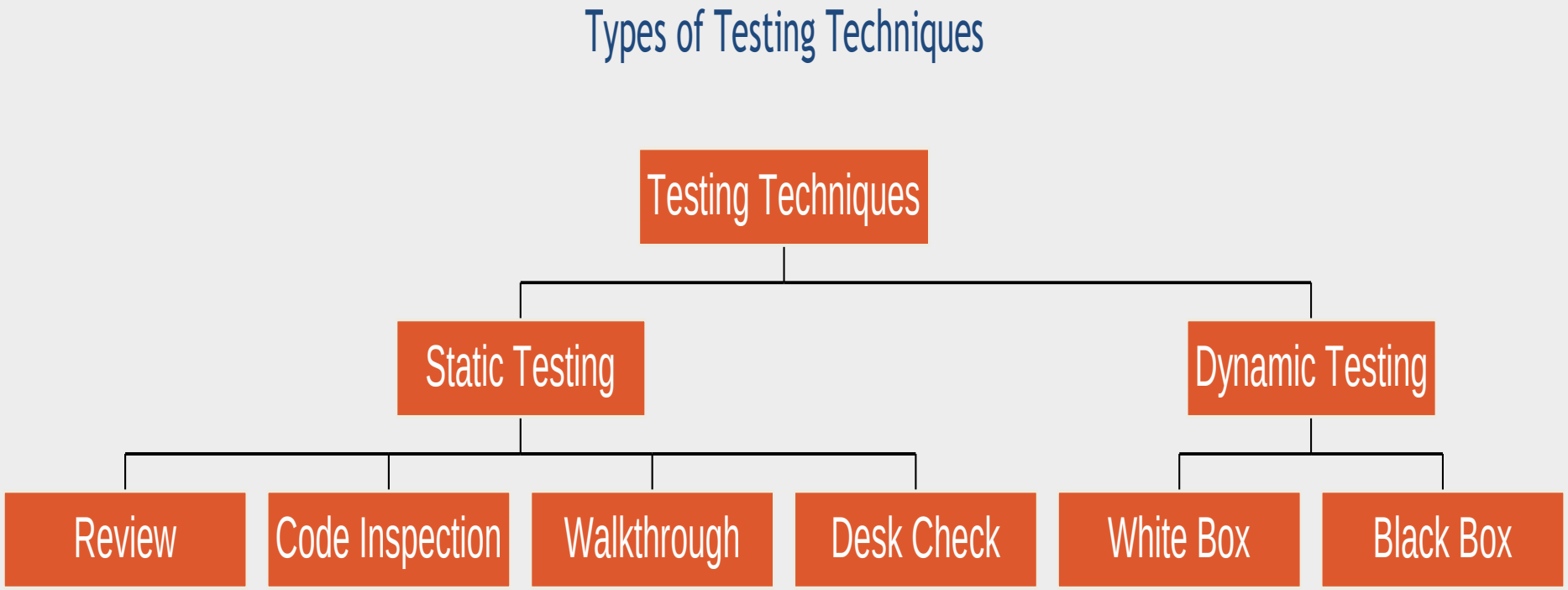
1. Static Testing

- It is a ***verification*** process
- Testing a software without execution on a computer. Involves just examination/review and evaluation
- It is done to test that software confirms to its SRS i.e. user specified requirements
- It is done for ***preventing*** the defects

2. Dynamic Testing

- It is a ***validation*** process
- Testing software through executing it
- It is done to test that software does what the user really requires
- It is done for ***detecting*** the defects

Static & Dynamic Testing Techniques



Differences between Static and Dynamic Testing

Static Testing	Dynamic Testing
It is the process of confirming whether the software meets its requirement specification.	It is the process of confirming whether the software meets user requirements.
Examples : Inspections, walkthroughs and reviews.	Examples : structural testing, black-box testing, integration testing, acceptance testing.
It is the process of inspecting without executing on computer.	It is the process of testing by executing on computer.
It is conducted to prevent defects.	It is conducted to correct the defects
It can be done before compilation	It takes place only after compilation and linking.

3.1 Static Testing Basics

- static testing relies on the manual examination of work products (i.e., reviews) or tool-driven evaluation of the code or other work products (i.e., static analysis).
- Both types of static testing assess the code or other work product being tested without actually executing the code or work product being tested.
- Static analysis is important for safety-critical computer systems (e.g., aviation, medical, or nuclear software).
- Static analysis is an important part of security testing.
- Static analysis is also often incorporated into automated build and delivery systems, for example in Agile development, continuous delivery, and continuous deployment.

3.1.1 Work Products examined by Static Testing

- Specifications, including business requirements, functional requirements, and security requirements
- Epics, user stories, and acceptance criteria
- Architecture and design specifications
- Code
- Testware, including test plans, test cases, test procedures, and automated test scripts
- User guides
- Web pages
- Contracts, project plans, schedules, and budgets
- Models, such as activity diagrams, which may be used for Model-Based testing

3.1.2 Benefits of Static Testing

- Detecting and correcting defects more efficiently and prior to dynamic testing.
- Identifying defects which are not easily found by dynamic testing
- Preventing defects in design or coding by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies in requirements
- Increasing development productivity (improved design, more maintainable code)
- Reducing development cost and time
- Reducing testing cost and time
- Reducing total cost of quality over the software's lifetime, due to fewer failures later in the lifecycle or after delivery into operation
- Improves communication between team members.
- Increased awareness of quality issues and early feedback on quality.

Typical Defects found during Static Testing

- Requirement defects (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies)
- Design defects (e.g., inefficient algorithms or database structures, high coupling, low cohesion)
- Coding defects (e.g., variables with undefined values, variables that are declared but never used, unreachable code, duplicate code)
- Deviations from standards (e.g., lack of adherence to coding standards)
- Incorrect interface specifications (e.g., different units of measurement used by the calling system than by the called system)
- Security vulnerabilities (e.g., susceptibility to buffer overflows)
- Gaps or inaccuracies in test basis traceability or coverage (e.g., missing tests for an acceptance criterion)

3.2 Review Process

Reviews vary from informal to formal.

- Informal reviews are characterized by not following a defined process and not having formal documented output.
- Formal reviews are characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the SDLC model, the maturity of the development process, the complexity of the work product to be reviewed, any legal or regulatory requirements, and/or the need for an audit trail.
- The focus of a review depends on the agreed objectives of the review (e.g., finding defects, gaining understanding, educating participants such as testers and new team members, or discussing and deciding by consensus).

3.2.1 Work Product Review Process

- The review process comprises the following main activities:
- Planning
- Initiate review
- Individual review (i.e., individual preparation)
- Issue communication and analysis
- Fixing and Reporting

Code Review Checklist used in Review Process

Data Reference Errors
- Is a variable referenced whose value is unset or uninitialized?

Data Declaration Errors
- Have all variables been explicitly declared?
- Are variables properly initialized in declaration sections?

Computation errors
- Are there any computations using variables having inconsistent data types?
- Is there any mixed mode computations?

Comparison errors
- Are there any comparisons between variables having inconsistent data types?

Control Flow errors
- Will every loop eventually terminate?
- Is it possible that, because of condition upon entry, a loop will never execute?

Interface errors
- Does the number of parameters received by these module equals the number of arguments sent by calling modules?
- Also is the order correct?

Input/output errors
- All I/O conditions handled correctly?

3.2.2 Roles & Responsibilities in a Formal Review

- **Author**
  - Creates the work product under review
  - Fixes defects in the work product under review (if necessary)
- **Management**
  - Is responsible for review planning
  - Decides on the execution of reviews
  - Assigns staff, budget, and time
  - Monitors ongoing cost-effectiveness
  - Executes control decisions in the event of inadequate outcomes
- **Facilitator (often called moderator)**
  - Ensures effective running of review meetings (when held)
  - Mediates, if necessary, between the various points of view
  - Is often the person upon whom the success of the review depends

3.2.2 Roles & Responsibilities in a Formal Review (Cont..)

- **Review leader**
  - Takes overall responsibility for the review
  - Decides who will be involved and organizes when and where it will take place
- **Reviewers**
  - May be SME, persons working on the project, stakeholders with an interest in the work product, and/or individuals with specific technical or business backgrounds
  - Identify potential defects in the work product under review
  - May represent different perspectives (e.g., tester, programmer, user, operator, business analyst, usability expert, etc.)
- **Scribe (or recorder)**
  - Collates potential defects found during the individual review activity
  - Records new potential defects, open points, and decisions from the review meeting (when held)

3.2.3 Review Types

- **Self Review**
- **Informal Review**
- **Walkthrough**
- **Technical Review**
- **Inspection**

Self Review

- Self review is done by the person who is responsible for a particular program code
- It is more of reviewing the code in informal way
- It is more like who writes the code, understands it better
- Self review is to be done by the programmer when he builds a new code
- There are review checklists that helps programmer to verify with the common errors regarding the program code

Informal Review

- Examples: buddy check, pairing, pair review
- Main purpose: detecting potential defects
- Possible additional purposes: generating new ideas or solutions, quickly solving minor problems
- Not based on a formal (documented) process
- May not involve a review meeting
- May be performed by a colleague of the author (buddy check) or by more people
- Results may be documented
- Varies in usefulness depending on the reviewers
- Use of checklists is optional
- Very commonly used in Agile development

Code Walkthrough

- Main purposes: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications
- Possible additional purposes: exchanging ideas about techniques or style variations, training of participants, achieving consensus
- Individual preparation before the review meeting is optional
- Review meeting is typically led by the author of the work product
- Scribe is mandatory
- Use of checklists is optional
- May take the form of scenarios, dry runs, or simulations
- Potential defect logs and review reports may be produced
- May vary in practice from quite informal to very formal

Technical Review

- Main purposes: gaining consensus, detecting potential defects
- Other purposes: evaluating quality and building confidence in the work product, generating new ideas, motivating and enabling authors to improve future work products, considering alternative implementations
- Reviewers should be technical peers of the author, and technical experts in the same or other disciplines
- Individual preparation before the review meeting is required
- Review meeting is optional, ideally led by a trained facilitator (not the author)
- Scribe is mandatory, ideally not the author
- Use of checklists is optional
- Potential defect logs and review reports are typically produced

Code Inspection

- Main purposes: detecting potential defects, evaluating quality and building confidence in the work product, preventing future similar defects through author learning and root cause analysis
- Possible further purposes: motivating and enabling authors to improve future work products and the software development process, achieving consensus
- Follows a defined process with formal documented outputs, based on rules and checklists
- An inspection team usually consists of clearly defined roles, such as those specified in section 3.2.2.

Code Inspection (Cont..)

- Individual preparation before the review meeting is required
- Reviewers are either peers of the author or experts in other disciplines that are relevant to the work product.
- Specified entry and exit criteria are used
- Scribe is mandatory
- Review meeting is led by a trained facilitator (not the author)
- Author cannot act as the review leader, reader, or scribe
- Potential defect logs and review report are produced
- Metrics are collected and used to improve the entire software development process, including the inspection process

Desk Checking (Peer Review)

- Human error detection technique
- Viewed as a one person inspection or walkthrough
- A person reads a program and checks it with respect to an error list and/or walks test data through it
- Less effective technique
- Best performed by the person other than the author of the program

3.2.4 Applying Review Techniques

There are a number of review techniques that can be applied during the individual review (i.e., individual preparation) activity to uncover defects.

- **Ad hoc**
- **Checklist-based**
- **Scenarios and dry runs**
- **Role-based**
- **Perspective-based**

3.2.5 Success Factors for Reviews

Organizational Success Factors for Reviews

- Each review has clear objectives, defined during review planning, and used as measurable exit criteria
- Review types are applied which are suitable to achieve the objectives and are appropriate to the type and level of software work products and participants
- Any review techniques used, such as checklist-based or role-based reviewing, are suitable for effective defect identification in the work product to be reviewed
- Any checklists used address the main risks and are up to date
- Large documents are written and reviewed in small chunks, so that quality control is exercised by providing authors early and frequent feedback on defects
- Participants have adequate time to prepare
- Reviews are scheduled with adequate notice
- Management supports the review process (e.g., by incorporating adequate time for review activities in project schedules)

3.2.5 Success Factors for Reviews (Cont..)

People-related success factors for reviews include:

- The right people are involved to meet the review objectives, for example, people with different skill sets or perspectives, who may use the document as a work input
- Testers are seen as valued reviewers who contribute to the review and learn about the work product, which enables them to prepare more effective tests, and to prepare those tests earlier
- Participants dedicate adequate time and attention to detail
- Reviews are conducted on small chunks, so that reviewers do not lose concentration during individual review and/or the review meeting (when held)

3.2.5 Success Factors for Reviews (Cont..)

People-related success factors for reviews include:

- Defects found are acknowledged, appreciated, and handled objectively
- The meeting is well-managed, so that participants consider it a valuable use of their time
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants
- Participants avoid body language and behaviors that might indicate boredom, exasperation, or hostility to other participants
- Adequate training is provided, especially for more formal review types such as inspections
- A culture of learning and process improvement is promoted

Summary

In this lesson, you have learnt:

- Basics of Static Testing
- Review Process
- Various Review Types
- Applying Review Techniques
- Success Factors for Reviews

Review Question

Question 1: _____ testing can discover dead codes

Question 2: The objective of walkthrough is to find errors but not solutions. (T/F)

Question 3: State the various Review Types

Question 4: State the Success factors for Review process

Testing Concepts

Lesson 4: Test Techniques

Lesson Objectives

To understand the following topics:

- Categories of Test Techniques
  - Choosing Test Techniques
  - Categories of Test Techniques & their Characteristics
- Black-box Test Techniques
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Decision Table Testing
  - State Transition Testing
  - Use Case Testing

Lesson Objectives

- White-box Test Techniques
  - Statement Testing and Coverage
  - Decision Testing and Coverage
  - The Value of Statement and Decision Testing
- Experience-based Test Techniques
  - Error Guessing
  - Exploratory Testing
  - Checklist-based Testing

4.1 Categories of Dynamic Test Techniques

- Dynamic Testing involves working with the software, giving input values and validating the output with the expected outcome
- Dynamic Testing is performed by executing the code
- It checks for functional behavior of software system , memory/CPU usage and overall performance of the system
- Dynamic Testing focuses on whether the software product works in conformance with the business requirements
- Dynamic testing is performed at all levels of testing and it can be either black or white box testing

Categories of Dynamic Test Techniques (Cont..)

White Box Test Techniques

- Code Coverage
  - Statement Coverage
  - Decision Coverage
  - Condition Coverage
  - Loop Testing
- Code complexity
  - Cyclomatic Complexity
- Memory Leakage

Black Box Test Techniques

- Equivalence Partitioning
- Boundary Value Analysis
- Use Case / UML
- Error Guessing
- Cause-Effect Graphing
- State Transition Testing

4.1.1 Choosing Test Techniques

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

- Type of component or system
- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels & Risk types
- Test objectives
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- SDLC model
- Expected use of the software
- Previous experience with using the test techniques on the component or system to be tested
- The types of defects expected in the component or system

4.1.2 Categories of Test Techniques & their Characteristics

Black-box test techniques

- It is a.k.a. behavioral/behavior-based techniques are based on an analysis of the appropriate test basis (e.g., formal requirements documents, specifications, use cases, user stories, or business processes).
- These concentrate on the inputs and outputs of the test object without reference to its internal structure.
- These techniques are applicable to both functional and nonfunctional testing.

White-box test techniques

- It is a.k.a. structural/structure-based techniques are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object.
- These concentrate on the structure and processing within the test object.

Experience-based test techniques

- These leverage the experience of developers, testers and users to design, implement, and execute tests.
- These techniques are often combined with black-box and white-box test techniques.

Characteristics of Black-box Test Techniques

- Test conditions, test cases, and test data are derived from a test basis that may include software requirements, specifications, use cases, and user stories
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis

Characteristics of White-box Test Techniques

- Test conditions, test cases, and test data are derived from a test basis that may include code, software architecture, detailed design, or any other source of information regarding the structure of the software
- Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces)
- Specifications are often used as an additional source of information to determine the expected outcome of test cases

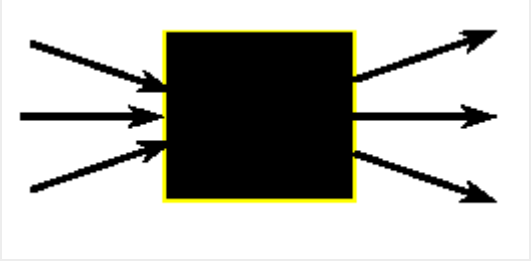
Characteristics of Experience-box Test Techniques

- Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders.
- This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects

4.2 Black Box Test Techniques

- Black box is data-driven, or input/output-driven testing
- The Test Engineer is completely unconcerned about the internal behavior and structure of program
- Black box testing is also known as behavioral, functional, opaque-box and closed-box
- Black Box can be applied at different Test Levels – Unit, Subsystem and System.

Input



Output

Black Box Test Techniques

There are various techniques to perform Black box testing ;

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State transition testing
- Use Case Testing
- Error Guessing

4.2.1 Equivalence Partitioning

- Equivalence partitioning divides data into partitions called as equivalence classes in such a way that all the members of a given partition are expected to be processed in the same way.
- There are equivalence partitions for both valid and invalid values.
  - Valid values are values accepted by the component or system. An equivalence partition containing valid values is called a "valid equivalence partition."
  - Invalid values are values rejected by the component or system. An equivalence partition containing invalid values is called an "invalid equivalence partition."
- Partitions can be identified for any data element related to the test object, including inputs, outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing).
- Assumption: If one value in a group works, all will work. One from each partition is better than all from one.

Guidelines & Examples to identify Equivalence Classes

1. If an input condition specifies a continuous range of values, there is one valid class, and two invalid classes

   E.g. The valid range of a mortgage applicant's income is $1000 - $75,000
   Valid class: {1000 > = income < = 75,000}
   Invalid classes: {income < 1000}, {income > 75,000}

2. If an input condition specifies a set of values, there is reason to believe that each is handled differently in the program.

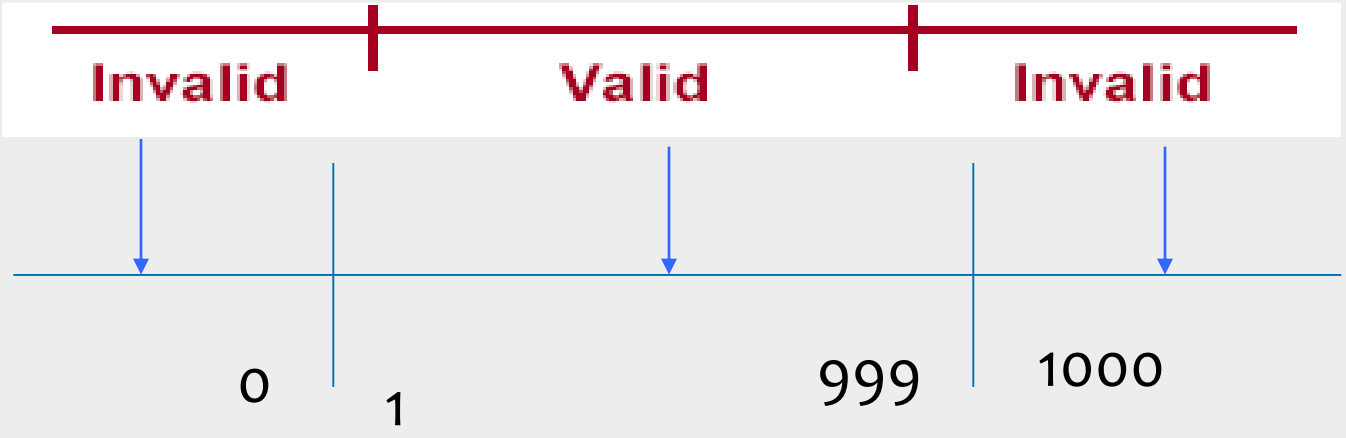
   E.g. Type of Vehicle must be Bus, Truck, Taxi). A valid equivalence class would be any one of the values and invalid class would be say Trailer or Van.

3. If a "must be" condition is required, there is one valid equivalence class and one invalid class

   E.g. The mortgage applicant must be a person
   Valid class: {person}
   Invalid classes:{corporation, ...anything else...}

Examples: Equivalence Partitioning

If an input condition specifies that a variable, say count, can take range of values(1 - 999). There is **one valid equivalence class (1 < count < 999) and two invalid equivalence classes (count < 1) & (count >999)**



4.2.2 Boundary Value Analysis

"Bugs lurk in corners and congregate at boundaries ….." *Boris Beizer*

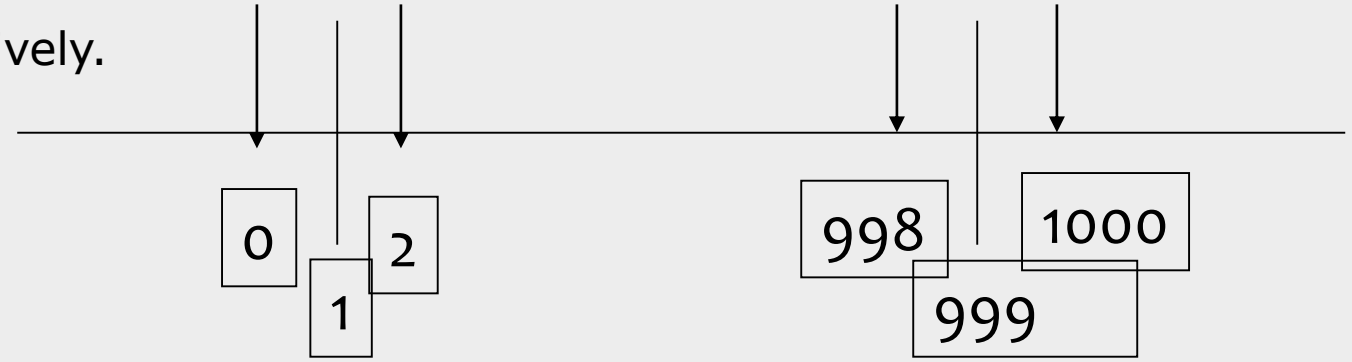
Boundary Conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes.

Boundary value analysis is a test case design technique that complements Equivalence partitioning but can only be used when the partition is ordered, consisting of numeric or sequential data. The minimum and maximum values (or first and last values) of a partition are its boundary values.

Test cases at the boundary of each input Includes the values at the boundary, just below the boundary and just above the boundary.

Guidelines & Examples for Boundary Value Analysis

E.g. From previous example, we have valid equivalence class as (1 < count < 999). Now, according to boundary value analysis, we need to write test cases for count=0, count=1,count=2,count=998,count=999 and count=1000 respectively.



E.g. If we have to test the function int Max(int a , int b) the Boundary Values for the arguments of the functions will be :

Arguments	Valid Values	Invalid Values
a	-32768, -32767, 32767, 32766	-32769,32768
b	-32768, -32767, 32767, 32766	-32769,32768

4.2.3 Decision Table Testing

- Decision Testing is useful for testing the implementation of system requirements that specify how different combinations of conditions result in different outcomes.
- When creating decision tables, the tester identifies conditions (often inputs) and the resulting actions (often outputs) of the system.
- These form the rows of the table, usually with the conditions at the top and the actions at the bottom.
  - Each column corresponds to a decision rule that defines a unique combination of conditions which results in the execution of the actions associated with that rule.
  - The values of the conditions and actions are usually shown as Boolean values (true or false) or can also be numbers or ranges of numbers.

Notations in Decision Tables

The common notation in decision tables is as follows:

For conditions:

- Y means the condition is true (may also be shown as T or 1)
- N means the condition is false (may also be shown as F or 0)
- — means the value of the condition doesn’t matter (may also be shown as N/A)

For actions:

- X means the action should occur (may also be shown as Y or T or 1)
- Blank means the action should not occur (may also be shown as – or N or F or 0)

Example of Decision Table

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Advantages of Decision Table Testing

1. The strength of decision table testing is that it helps to identify all the important combinations of conditions, some of which might otherwise be overlooked.
2. It also helps in finding any gaps in the requirements.
3. It may be applied to all situations in which the behavior of the software depends on a combination of conditions, at any test level.

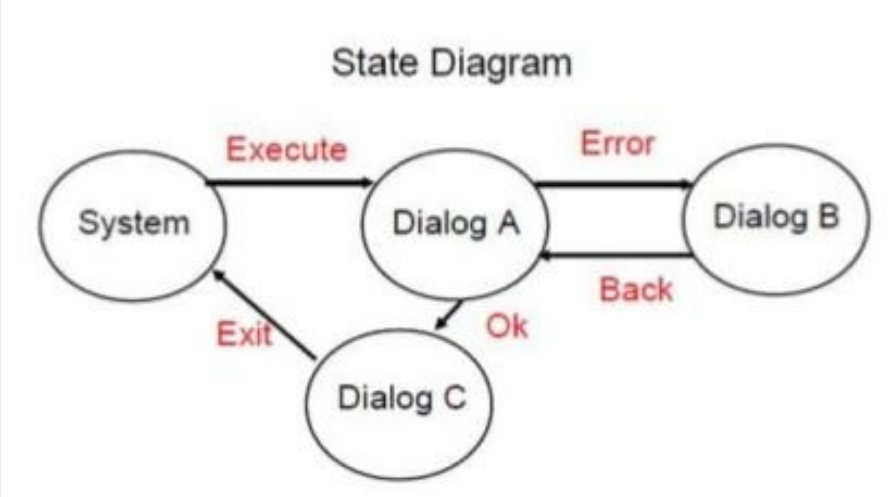
4.2.4 State Transition Testing

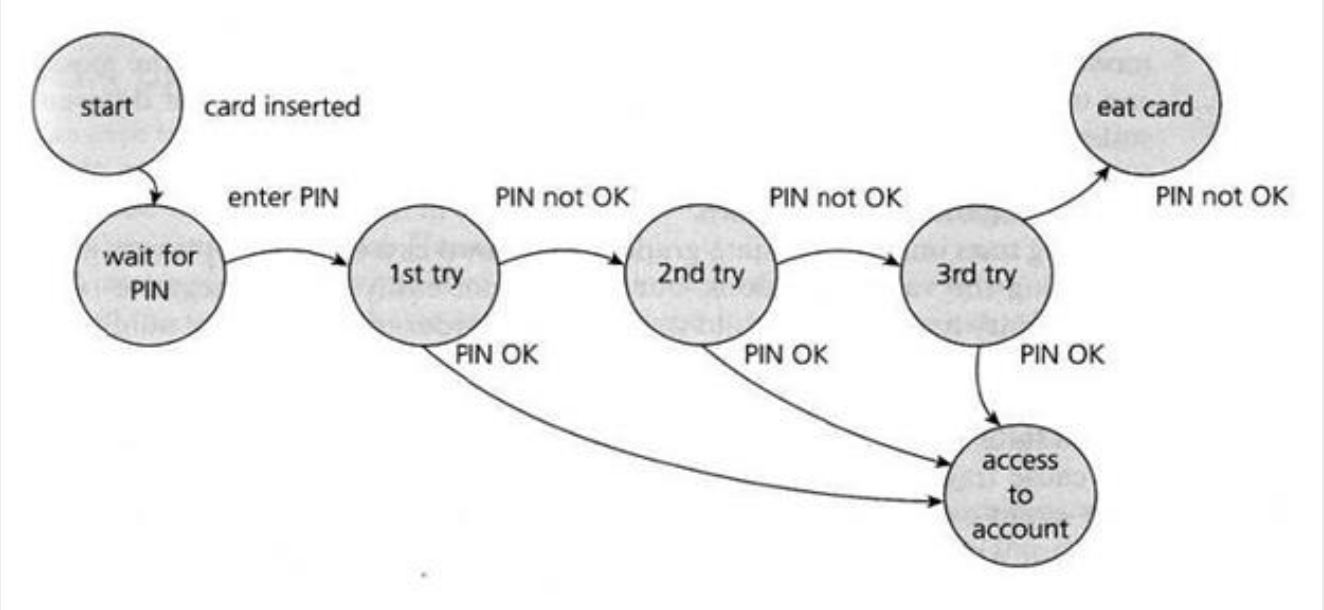
- A state transition diagram shows the possible software states, as well as how the software enters, exits, and transitions between states.
- A transition is initiated by an event (e.g., user input of a value into a field). The event results in a transition.
- If the same event can result in two or more different transitions from the same state, that event may be qualified by a guard condition.
- The state change may result in the software taking an action (e.g., outputting a calculation or error message).

Example :

- The State Transition testing is used for Menu-based application.
- The program starts with an introductory menu. As an option is selected the program changes state and displays a new menu. Eventually it displays some information , data input screen.
- Each option in each menu should be tested to validate that each selection made takes us to the state we should reach next.

Guidelines & Examples of State Transition Testing





4.2.5 Use Case Testing

- Tests can be derived from use cases, which are a specific way of designing interactions with software items, incorporating requirements for the software functions represented by the use cases.
- Use cases are associated with actors (human users, external hardware, or other components or systems) and subjects (the component or system to which the use case is applied).
- Each use case specifies some behavior that a subject can perform in collaboration with one or more actors.
- A use case can be described by interactions and activities, as well as preconditions, post conditions and natural language where appropriate.
- Interactions between the actors and the subject may result in changes to the state of the subject.

4.3 White Box Test Techniques

- White box is logic driven testing and permits Test Engineer to examine the internal structure of the program
- Examine paths in the implementation
- Make sure that each statement, decision branch, or path is tested with at least one test case.
- Desirable to use tools to analyze and track Coverage
- White box testing is also known as structural, glass-box and clear-box

There are various techniques to perform Black box testing ;

- Code Coverage
  - Statement Testing and Coverage
  - Decision Testing and Coverage
  - Condition Testing and Coverage
- Code complexity

4.3.1 Statement Coverage

Test cases must be such that all statements in the program is traversed at least once.

Example :

Consider the following snippet of code

```
void procedure(int a, int b, int x)
{
    If (a>1) && (b==0)
            { x=x/a;  }   //statement 1
     If (a==2) || (x>1)
            {  x=x+1; }  //statement 2
}
```

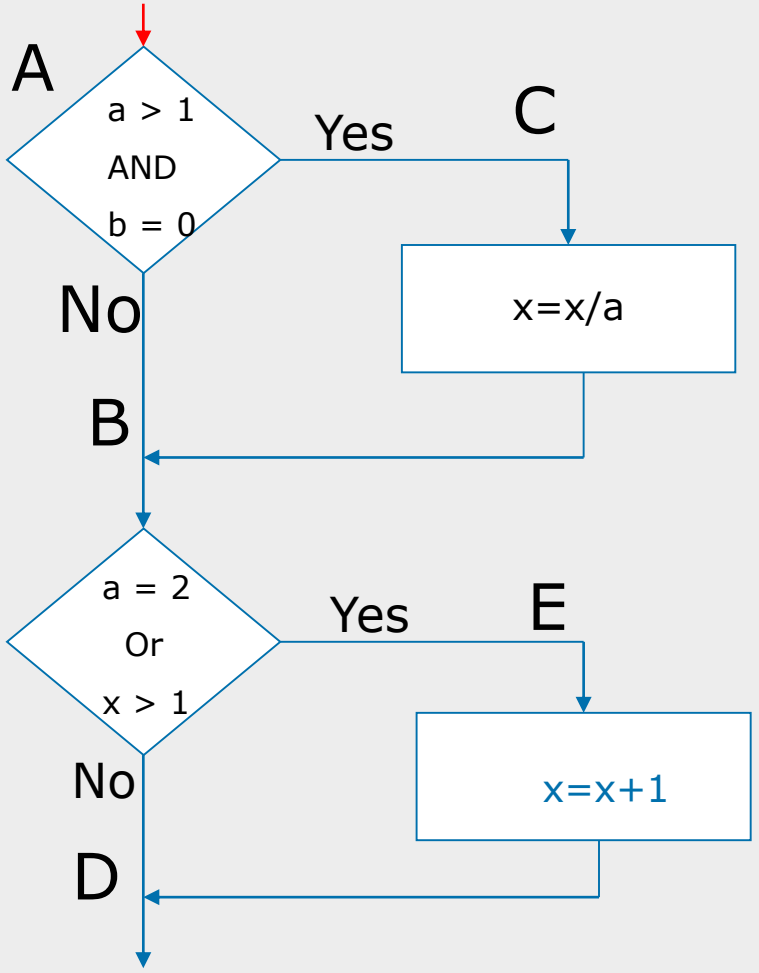
Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

Statement Coverage

Test Case 1: a=2,b=0, x=3.

Every statement will be executed once.

One test case is sufficient to execute all the statements in the code.

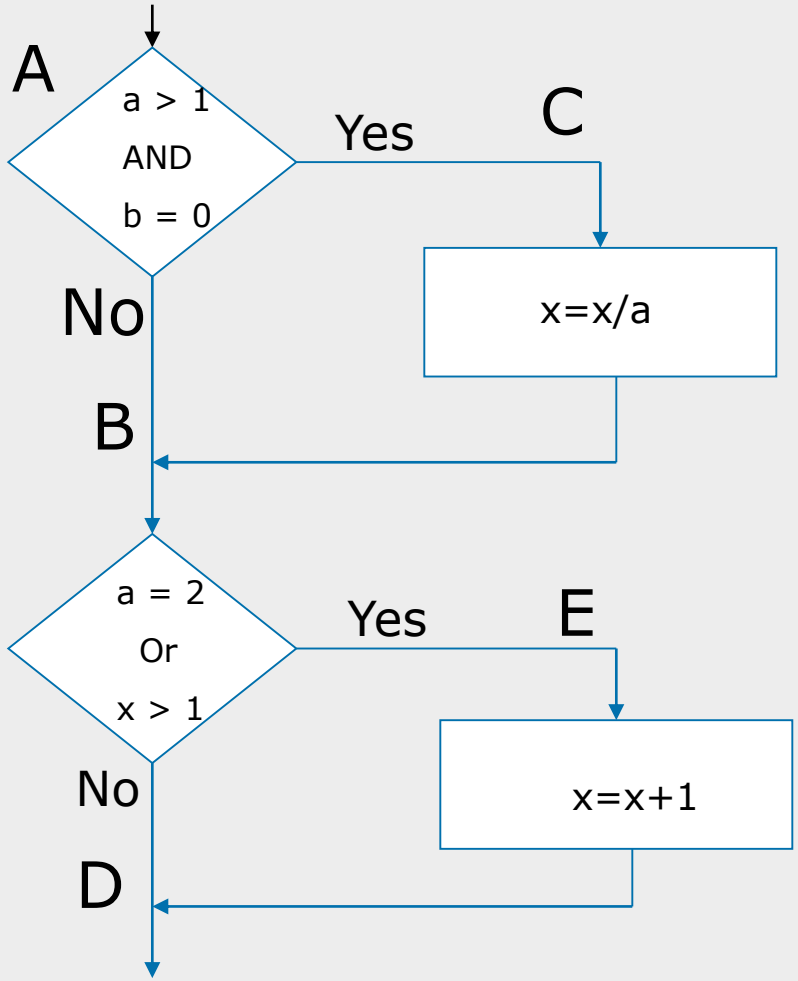


4.3.2 Decision Coverage

Test Case 1: a=2, b=0, x>1
(Decision1 is True, Decision2 is True) (Path ACE)

Test Case 2: a<=1 , b!=0, x<=1
(Decision1 is False, Decision2 is False) (Path ABD).

Two test cases are sufficient to test all decisions – every decision should be tested at least once for both TRUE and FALSE sides.

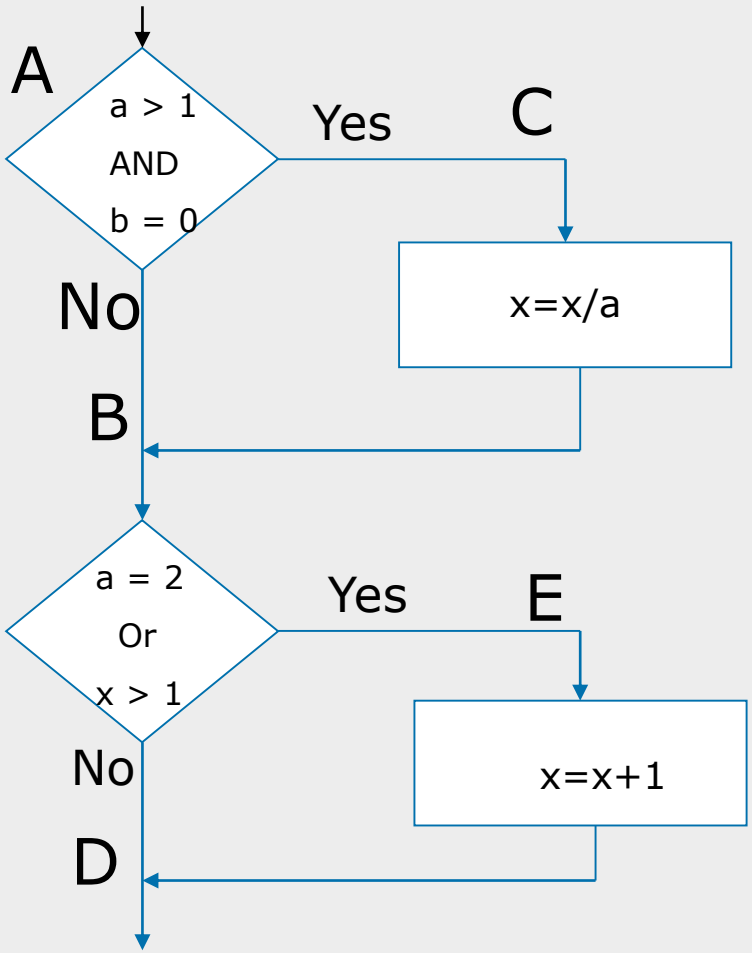


4.3.3 Value of Statement and Decision Testing

- When 100% statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Statement testing provides less coverage than decision testing.
- When 100% decision coverage is achieved, it executes all decision outcomes, which includes testing the true outcome and also the false outcome, even when there is no explicit false statement (e.g., in the case of an IF statement without an else in the code).
- Statement coverage helps to find defects in code that was not exercised by other tests. Decision coverage helps to find defects in code where other tests have not taken both true and false outcomes.
- **Achieving 100% decision coverage guarantees 100% statement coverage.**

4.3.4 Condition Coverage

Test cases are written such that each condition in a decision takes on all possible outcomes at least once.



Test Case1 : a=2, b=0, x=3 (Condition1 is True,Condn2 is True)

(Path ACE)

Test Case2 : a=3, b=0, x=0

(Condn1 is True,Condn2 is False,Condn3 is False)

(Path ACD)

Test Case3 : a=1, b=0, x=3

(Condition1 is False,Condition2 is True)

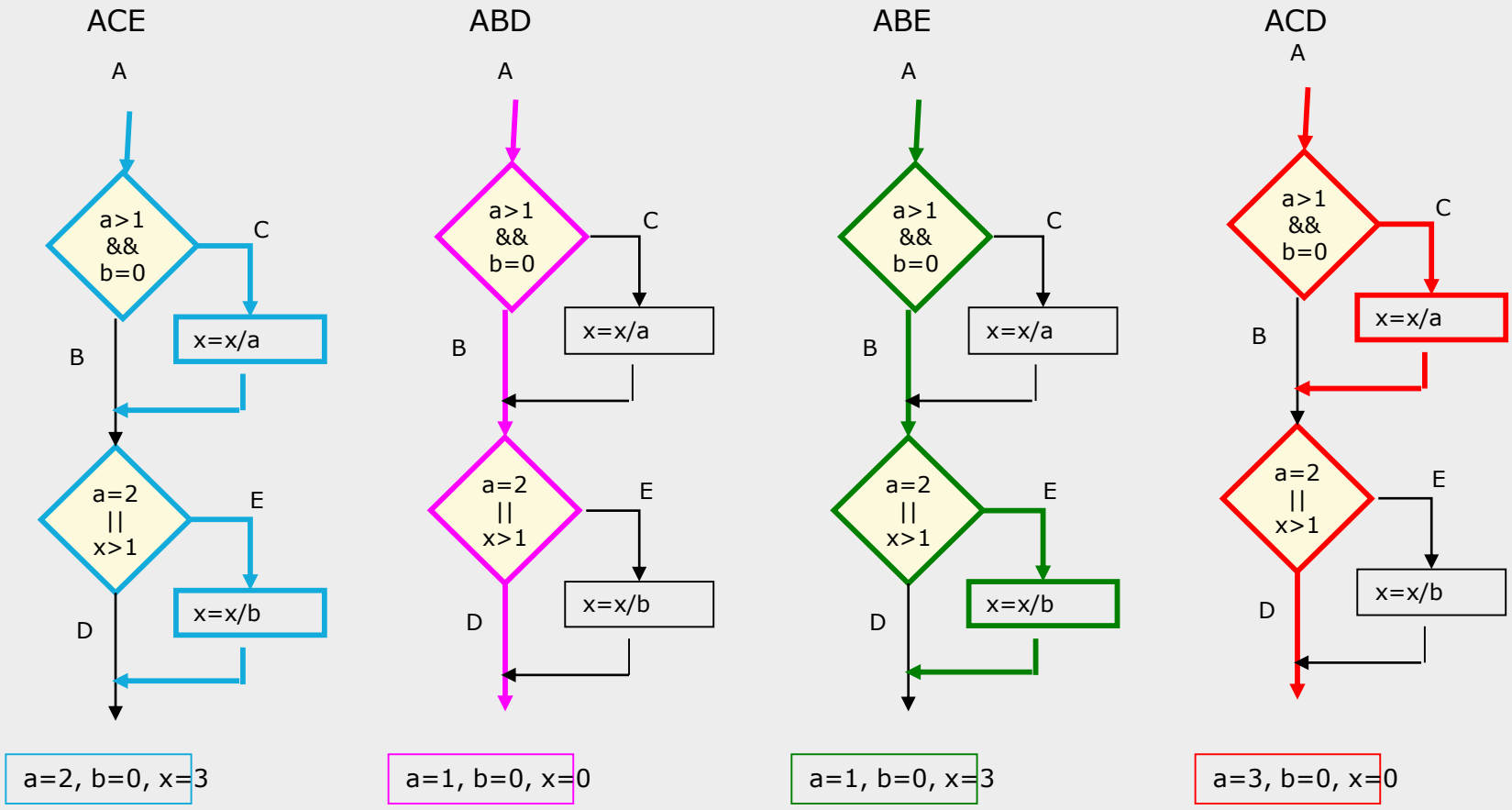
(Path ABE)

Test Case4: a=1, b=1, x=1

(Condition1 is False,Condition2 is False)

(Path ABD)

Condition Coverage



Cyclomatic Complexity

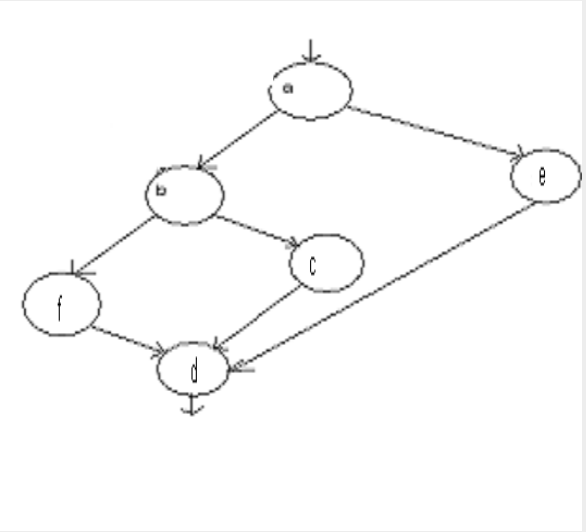
- Cyclomatic Complexity (Code Complexity) is a software metric that provides a quantitative measure of logical complexity of a program
- When Used in the context of the basis path testing method, value for cyclomatic complexity defines number of independent paths in basis set of a program
- Also provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once
- Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity

Calculating Cyclomatic Complexity

- The cyclomatic complexity of a software module is calculated from a flow graph of the module , when used in context of the basis path testing method
- Cyclomatic Complexity V(G) is calculated one of the three ways:
  - V(G) = E - N + 2 , where E is the number of edges and N = the number of nodes of the graph
  - V(G) = P+1, where P is the number of predicate nodes
  - V(G) = R , where number of region in the graph

Calculating Cyclomatic Complexity : Example

In the given figure a and b are predicate nodes



1. Cyclomatic Complexity, V(G) for a flow Graph G is V(G) = E – N + 2
   E = Number of Edges in the graph (7 in the above figure)
   N = number of flow graph Nodes (6)
   R = number of Regions (3)
   Hence V(G) = 7-6+2 = 3
2. V(G) can also be calculated as V(G) = P+1, where P is the number of predicate nodes. Here V(G) = 2+1 = 3
3. Also V(G) can be calculated as V(G) = R hence V(G) = 3

4.4 Experience-based Test Techniques

- When applying experience-based test techniques, the test cases are derived from the tester's skill and intuition, and their experience with similar applications and technologies.
- These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques.
- Depending on the tester's approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness.
- Coverage can be difficult to assess and may not be measurable with these techniques.

Commonly used experience-based techniques are

- Error Guessing
- Exploratory Testing
- Checklist-based Testing

4.4.1 Error Guessing

It is an ad hoc approach

Error guessing is a technique used to anticipate the occurrence of mistakes, defects, and failures, based on the tester's knowledge, including:

- How the application has worked in the past
- What types of mistakes the developers tend to make
- Failures that have occurred in other applications

A methodical approach to the error guessing technique is to create a list of possible mistakes, defects, and failures, and design tests that will expose those failures and the defects that caused them.

Example :

Suppose we have to test the login screen of an application. An experienced test engineer may immediately see if the password typed in the password field can be copied to a text field which may cause a breach in the security of the application.

4.4.2 Exploratory Testing

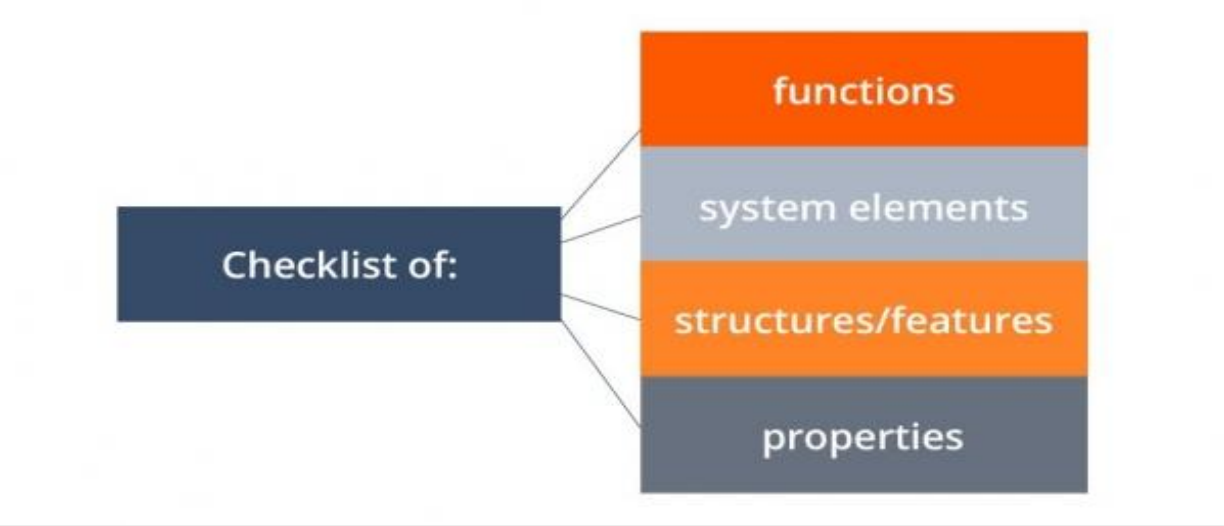
- In exploratory testing, informal (not pre-defined) tests are designed, executed, logged, and evaluated dynamically during test execution.
- The test results are used to learn more about the component or system, and to create tests for the areas that may need more testing.
- Exploratory testing is sometimes conducted using session-based testing to structure the activity.
- Exploratory testing is most useful when there are few or inadequate specifications or significant time pressure on testing.
- Exploratory testing is also useful to complement other more formal testing techniques. Exploratory testing is strongly associated with reactive test strategies (see section 5.2.2). Exploratory testing can incorporate the use of other black-box, white-box, and experience-based techniques.

4.4.3 Checklist-based Testing

- In checklist-based testing, testers design, implement, and execute tests to cover test conditions found in a checklist.
- As part of analysis, testers create a new checklist or expand an existing checklist, but testers may also use an existing checklist without modification.
- Such checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.
- Checklists can be created to support various test types, including functional and non-functional testing.
- In the absence of detailed test cases, checklist-based testing can provide guidelines and a degree of consistency.
- As these are high-level lists, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

Check-list based Technique

Commonly used Testing Checklists



Advantages of using checklists in testing

- Flexibility.
- Easy to create.
- Analyzing the results.
- Team integration.
- Deadlines control.
- Test case reusability can help in cutting down costs incurred in missing out on important testing aspects.
- innovative testing strategy can be added in the testing checklist.

Difficulties of using checklists in testing

- Different interpretation
- Difficulty in Test results reproducibility
- “Holes” in coverage.
- Item overlap.
- Reporting problems.

Summary

In this lesson, you have learnt:

- The test case techniques discussed so far need to be combined to form overall strategy
- Each technique contributes a set of useful test cases, but none of them by itself contributes a thorough set of test cases

Review Question

Question 1: For calculating cyclomatic complexity, flow graph is mapped into corresponding flow chart

- Option: True / False

Question 2: How many minimum test cases required to test a simple loop?

Question 3: Incorrect form of logic coverage is :

- Statement coverage
- Pole coverage
- Condition coverage
- Path coverage

Question 4: One test condition will have _____ test cases.

Review Question

Question 5: For Agile development model conventional testing approach is followed.

- Option: True / False

Question 6: A test case is a set of _____, _____, and _____ developed for a particular objective.

Question 7: An input field takes the year of birth between 1900 and 2004. State the boundary values for testing this field.

- 0, 1900,2004,2005
- 1900, 2004
- 1899, 1900, 2004, 2005
- 1899, 1900, 1901, 2003, 2004, 2005

Review Question: Match the Following

1. Code coverage	A. Flow graph
2. Interface errors	B. Loop testing
3. Code complexity	C. Black box testing
	D. Flow chart
	E. Condition testing
	F. White box testing

Testing Concepts

Lesson 5: Test Management & Test Metrics

Lesson Objectives

To understand the following topics

- Test Organization
  - Independent Testing
  - Tasks of a Test Manager and Tester
- Test Planning and Estimation
  - Purpose and Content of a Test Plan
  - Test Strategy and Test Approach
  - Entry Criteria and Exit Criteria (Ready and Done)
  - Test Execution Schedule
  - Factors Influencing the Test Effort
  - Test Estimation Techniques
- Test Monitoring and Control
  - Metrics Used in Testing
  - Purposes, Contents, and Audiences for Test Reports
- Configuration Management
- Risks and Testing
  - Definition of Risk
  - Product and Project Risks
  - Risk-based Testing and Product Quality
- Defect Management

5.1.1 Independent Testing

- Independent testing brings a perspective which is different than that of the authors since they have different cognitive biases from the authors.
- Unbiased testing is necessary to objectively evaluate quality of a software
- Developer carrying out testing would not like to expose defects
- Assumptions made are carried into testing
- People see what they want to see.
- More effective in terms of Quality & Cost
- It is conducted by an independent test team other than developer to avoid author bias and is more effective in finding defects and failures
- The tester sees each defect in a neutral perspective
- The tester is totally unbiased
- The tester sees what has been built rather than what the developer thought
- The tester makes no assumptions regarding quality

5.1.1 Independent Testing (Cont..)

Benefits of Test Independence :

- Independent testers are likely to recognize different kinds of failures compared to developers because of their different backgrounds, technical perspectives, and biases.
- An independent tester can verify, challenge, or disprove assumptions made by stakeholders during specification and implementation of the system

Drawbacks of Test Independence :

- Isolation from the development team, leading to lack of collaboration, delays in providing feedback to the development team, or an adversarial relationship with the development team
- Developers may lose a sense of responsibility for quality
- Independent testers may be seen as a bottleneck or blamed for delays in release
- Independent testers may lack some important information about the test object)

5.2.1 Purpose and Content of a Test Plan

- A test plan outlines test activities for development and maintenance projects.

- Test Planning is influenced by the test policy and test strategy of the organization, the development lifecycles and methods being used, the scope of testing, objectives, risks, constraints, criticality, testability, and the availability of resources.

Test Plan Contents (IEEE 829)

1. Test Plan Identifier
2. References
3. Introduction
4. Test Items
5. Software Risk Issues
6. Features to be Tested
7. Features not to be Tested
8. Test Approach (Strategy)
9. Item Pass/Fail Criteria
10. Suspension Criteria and Resumption Requirements
11. Test Deliverables
12. Testing Tasks
13. Environmental Needs
14. Staffing and Training Needs
15. Responsibilities
16. Schedule
17. Planning Risks and Contingencies
18. Approvals
19. Glossary

Test Planning Activities

- Determining the scope, risks and identifying the objectives of testing
- Defining the overall approach of testing, definition of the test levels, entry and exit criteria
- Integrating and coordinating the testing activities into the SDLC activities
- Making decisions about what to test, what roles will perform the test activities, how the test activities should be done, and how the test results will be evaluated
- Scheduling test analysis and design activities
- Scheduling test implementation, execution and evaluation
- Assigning resources for the different activities
- Defining the amount, level of detail and templates for the test documentation
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues

5.2.2 Test Strategy and Test Approach

Test Strategy?

- A test strategy describes the test process at the product or organizational level.
- Test strategy means “How you are going to test the application?” You need to mention the exact process/strategy that you are going to follow when you will get the application for testing.

Test Approach:

- A test approach is the test strategy implementation of a project, defines how testing would be carried out

5.2.3 Entry Criteria (Ready) and Exit Criteria (Done)

Entry Criteria :

- Availability of testable requirements, user stories, and/or models (e.g., when following a model based testing strategy)
- Availability of test items that have met the exit criteria for any prior test levels
- Availability of test environment
- Availability of necessary test tools
- Availability of test data and other necessary resources

5.2.3 Entry Criteria and Exit Criteria (Cont..)

Exit Criteria :

- Planned tests have been executed
- A defined level of coverage (e.g., of requirements, user stories, acceptance criteria, risks, code) has been achieved
- The number of unresolved defects is within an agreed limit
- The number of estimated remaining defects is sufficiently low
- The evaluated levels of reliability, performance efficiency, usability, security, and other relevant quality characteristics are sufficient

Example : Entry Criteria for Functional Testing

- Integration Testing is complete and sign-off is received by Project team.
- Integration test results are provided to the QA team within the Integration Execution & Signoff artifact.
- Development team provides a demonstration of application changes prior to promotion to QA Environment
- Code is delivered and successfully promoted to the Functional/System Test Environment as described in Master Test Plan
- Functional/System Test planning is detailed, reviewed and approved within the Master Test Plan
- Smoke /Shake down test has been completed to ensure test environment is stable for testing.
- Functional/System Test Cases are created, reviewed and approved within the RBC Enterprise approved tool (HP QC)
- Test data is ready for Functional/System Testing

Example : Exit Criteria for Functional Testing

- All high and medium risk tests identified in the detailed test plan are executed, including interface testing
- All planned testing is complete and documented
- Functional/System test execution results are captured
- All known defects have been entered into the defect tracking tool
- There are no known severity one or severity two defects
- Action plans have been created for outstanding severity three and four defects
- Appropriate signoffs are obtained
- Location of test cases, automated test scripts, defects and Functional/System Execution & Signoff artefact are detailed within the SCM plan.
- Any known deviations from the BRD and SRS are documented and approved

5.2.4 Test Case Execution Schedule

Pre-execution activities

Setting up the Environment

- Similar to production environment
- Hardware (e.g. Hard Disk, RAM, Processor)
- Software (e.g. IE, MS office)
- Access to Applications

Setting up data for Execution

- Any format (e.g. xml test data, system test data, SQL test data)
- Create fresh set of your own test data
- Use existing sample test data
- Verify, if the test data is not corrupted
- Ideal test data - all the application errors get identified with minimum size of data set

Pre-execution Activities

Test data to ensure complete test coverage

Design test data considering following categories:

- No data
  - Relevant error messages are generated
- Valid data set
  - Functioning as per requirements
- Invalid data set
  - Behavior for negative values
- Boundary Condition data set
  - Identify application boundary cases
- Data set for Performance, Load and Stress Testing
  - This data set should be large in volume

Setting up the Test Environment

There are various types of Test Environments :

- Unit Test Environment
- Assembly/Integration Test Environment
- System/Functional/QA Test Environment
- User Acceptance Test Environment
- Production Environment

Before starting Execution

Validate the Test Bed

- Environment
  - Hardware (e.g. Hard Disk, RAM, Processor)
  - Software (e.g. IE, MS office)
- Access
  - Access to the Application
  - Availability of Interfaces (e.g. Printer)
  - Availability of created Test Data
- Application
  - High level testing on the application to verify if the basic functionality is working
  - There are no show-stoppers
  - Referred to as Smoke/Sanity/QA Build Acceptance testing

During Execution

Run Tests

- Run test on the identified Test Bed
- Precondition
- Use the relevant test data

Note the Result

- Objective of test case
- Action performed
- Expected outcome
- Actual outcome
- Pass/Fail (according to pass/fail criteria)

Compare the Input and Output

- Validate the data (e.g. complex scenarios, data from multiple interfaces)

Record the Execution

- Test data information (e.g. type of client, account type)
- Screenshots of the actions performed and results
- Video recording (HP QC Add-in)

After Execution

Report deviation

- Log Defect for Failed Test Cases
- Defect logging
  - Project
  - Summary
  - Description
  - Status
  - Detected By
  - Assigned To
  - Environment (OS, Release, Build, Server)
  - Severity
  - Priority
  - Steps to recreate and Screenshots

5.2.5 Factors Influencing the Test Effort

Product Characteristics :

- The risks associated with the product
- The quality of the test basis
- The size of the product
- The complexity of the problem domain
- The requirements for quality characteristics (e.g., security, reliability)
- The required level of detail for test documentation
- Requirements for legal and regulatory compliance

Development process Characteristics :

- The stability and maturity of the organization
- SDLC model used
- Tools used
- Test approach
- Test process
- Time pressure

5.2.5 Factors Influencing the Test Effort (Cont..)

People Characteristics :

- Skills and experience of the team members
- Team Cohesion and Leadership

Test Results:

- The number and severity of defects
- The amount of rework required

5.2.6 Test Estimation Techniques

There are several estimation techniques to determine the effort required for adequate testing. Two most used techniques are:

- The metrics-based technique: estimating the test effort based on metrics of former similar projects, or based on typical values
- The expert-based technique: estimating the test effort based on the experience of the owners of the testing tasks.

Example:

In Agile development, burndown charts are examples of metrics-based approach as effort is being captured and reported, and is then used to feed into the team's velocity to determine the amount of work the team can do in the next iteration; whereas planning poker is an example of the expert-based approach, as team members are estimating the effort to deliver a feature based on their experience.

5.3 Test Monitoring and Control

Why Test monitoring is necessary?

- To know the status of the testing project at any given point in time
- To provide visibility on the status of testing to other stake holders
- To be able to measure testing against defined exit criteria
- To be able to assess progress against Planned schedule & Budget

5.3 Test Monitoring and Control (Cont..)

Why Test Control is necessary?

- To guide and suggest corrective actions based on the information and metrics gathered and reported.
- Actions may cover test activity and may effect any other SDLC activity.
- Examples of test control actions include:
  - Re-prioritizing tests when an identified risk occurs (e.g., software delivered late)
  - Changing the test schedule due to availability or unavailability of a test environment or other resources
  - Re-evaluating whether a test item meets an entry or exit criterion due to rework

5.3.1 Metrics used in Testing

- Efficient test process measurement is essential for managing and evaluating the effectiveness of a test process. Test metrics are an important indicator of the effectiveness of a software testing process.

Metrics should be collected during and at the end of a test level in order to assess :

- Progress against the planned schedule and budget
- Current quality of the test object
- Adequacy of the test approach
- Effectiveness of the test activities with respect to the objectives

5.3.1 Metrics used in Testing (Cont..)

Common Test Metrics include:

- Percentage of planned work done in test case preparation (or percentage of planned test cases implemented)
- Percentage of planned work done in test environment preparation
- Test case execution (e.g., number of test cases run/not run, test cases passed/failed, and/or test conditions passed/failed)
- Defect information (e.g., defect density, defects found and fixed, failure rate, and confirmation test results)
- Test coverage of requirements, user stories, acceptance criteria, risks, or code
- Task completion, resource allocation and usage, and effort
- Cost of testing, including the cost compared to the benefit of finding the next defect or the cost compared to the benefit of running the next test

Need of Metrics

- To track Projects against plan
- To take timely corrective actions
- To get early warnings
- It is a basis for setting benchmarks
- It is a basis for driving process improvements
- To track the process performance against business

Types of Metrics

- **Project Metrics** - Test Coverage, Defect Density, Defect arrival rate

- **Process Metrics** - Test Effectiveness, Effort Variance, Schedule Variance, CoQ, Delivered Defect Rate, Defect Slippage or Test escape, Defect Injection Rate, Rejection Index, Resource Utilization, Review Effectiveness, Test Case Design, Rework Index, Defect Removal Efficiency.

- **Productivity Metrics** - Test case design productivity, Test case execution productivity

- **Closure Metrics** – Effort distribution metrics like Test Design Review Effort, Test Design Rework effort, KM Effort

Types of Metrics – Project Metrics

Test Coverage: The following are the test coverage metrics:

Test Design:

- # Of Requirements or # Of Use Cases covered / # Of Requirements or # Of Use Cases Planned

Test Execution:

- # Of Test scripts or Test cases executed/# Of Test scripts or Test cases Planned

Test Automation:

- # Of Test cases automated/# Of Test cases

Types of Metrics – Project Metrics (Cont..)

Defect Density

Total Defect density = (Total number of defects including both impact and non-impact, found in all the phases + Post delivery defects)/Size

Defect arrival rate:

Of Defects * 100 / # of Test Cases planned for Execution

This metric indicates the quality of the application/product under test.

Lower the value of this parameter is better.

Types of Metrics – Process Metrics

Test Effectiveness

Of Test Cases failed (found defects)/# Of Test Cases executed

This metric indicates the effectiveness of the Test Cases in finding the defects in the product

Defect Removal Efficiency

(# of Defects found internally / Total # Of(internal + external) Defects found) * 100

It indicates the number of defects leaked after several levels of review and these defects are slipped to the customer.

Defect Injection Rate (No of Defects / 100 Person Hours)

No of Defects[phase wise] * 100/ Actual Effort[phase wise]

This is used to detect the defects injected during STLC Phases.

Types of Metrics – Process Metrics (Cont..)

Cost of quality

% CoQ = (Total efforts spent on Prevention + Total efforts spent on Appraisal + Total efforts spent on failure or rework)*100/(Total efforts spent on project)

Prevention Cost: (Green Money) :

Cost of time spent by the team in implementing the preventive actions identified from project start date to till date

Appraisal Cost: (Blue Money) :

Cost of time spent on review and testing activities from the project start date to till date

Failure Cost: (Red Money) :

Cost of time taken to fix the pre and post delivery defects. Expenses incurred in rework – Customer does not pay for this

Types of Metrics – Process Metrics (Cont..)

Effort Variance

Overall and Variance at each milestone

(Actual effort - Planned effort) / Planned effort) * 100

The purpose of this parameter is to check the accuracy of the Effort estimation process to improve the estimation process.

Schedule variance

(Actual end date - Planned end date) / (Planned end date - Plan start date + 1) * 100

It depicts the ± buffer that can be used for optimum use of resource deployment and monitor the dates committed to the client and helps in better planning of future tasks.

Types of Metrics – Process Metrics (Cont..)

Defect slippage or Test escape

(Total # Of External Defects / Total # Of Defects detected (Internal+External)) * 100

This measure helps us to know how effectively we are detecting the defects at various stages of internal testing

Rejection index

Of Defects rejected/# Of Defects raised

The purpose of this parameter is to measure the Quality of the defects raised

Types of Metrics – Process Metrics (Cont..)

Resource Utilization

Actual effort utilized in the month for project activities /Total available Effort in the month

Review Effectiveness

(No of Internal Review Defects / [No of Internal Defects+No of External Defects])*100

The purpose of this parameter is to measure how effective are our reviews in capturing all the phase injected defects.

Test Case Design Rework Index

(Test Cases with Review Comments for rework/Total Test Cases)*100

Types of Metrics – Productivity Metrics

Test case design productivity

Of Test cases (scripts) designed/ Total Test case design effort in hours

This metric indicates the productivity of the team in designing the test cases.

Test case execution productivity

Of Test cases executed/ Total Test case executed effort in hours

Effort shall include the set up time, execution time. This metric indicates the productivity of the team in executing the test cases.

Types of Metrics – Closure Metrics

Test Design Review effort

(Effort spent on Test Case design reviews / Total effort spent on Test Case design) * 100

This is used with other process metrics like "Review Effectiveness", "DRE", "Defect Injection Ratio" to plan for an adequate review effort for future projects.

Test Design Rework effort

(Effort spent on Test Case design review rework / Total effort spent on Test Case design) * 100

This can be used with other process metrics like "Effort Variance", "Schedule Variance" to plan for an adequate rework effort for future projects.

KM Effort

(Total Effort spent on preparation of KM artifacts / Total efforts for entire project) * 100

This indicates effort spent on KM that can be used to plan KM activities for future projects.

5.3.2 Purposes, Contents & Audiences for Test Reports

- The purpose of test reporting is to summarize and communicate test activity information, both during and at the end of a test activity (e.g., a test level).
- The test report prepared during a test activity may be referred to as a **test progress report**, while a test report prepared at the end of a test activity may be referred to as a **test summary report (test completion report).**

Test Progress Report includes:

- The status of the test activities and progress against the test plan
- Factors impeding progress
- Testing planned for the next reporting period
- The quality of the test object

5.3.2 Purposes, Contents & Audiences for Test Reports (Cont..)

Test Summary Report includes:

- Summary of testing performed
- Information on what occurred during a test period
- Deviations from plan, including deviations in schedule, duration, or effort of test activities
- Status of testing and product quality with respect to the exit criteria or definition of done
- Factors that have blocked or continue to block progress
- Metrics of defects, test cases, test coverage, activity progress, and resource consumption.
- Residual risks
- Reusable test work products produced

5.3.2 Purposes, Contents & Audiences for Test Reports (Cont..)

The contents of a test report will vary depending on the project, the organizational requirements, and the software development lifecycle.

- Example : a complex project with many stakeholders or a regulated project may require more detailed and rigorous reporting than a quick software update.
- Example : in Agile development, test progress reporting may be incorporated into task boards, defect summaries, and burndown charts, which may be discussed during a daily stand-up meeting

5.4 Configuration Management & Configuration Control

Configuration Management :

- The purpose of configuration management is to establish and maintain the integrity of the component or system, the test ware, and their relationships to one another through the project and product lifecycle.
- A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item

Configuration Control or Version control:

- An element of configuration management, consisting of evaluation, coordination, approval or disapproval and implementation of changes to configuration items after formal establishment of their configuration identification

5.4 Configuration Management & Configuration Control (Cont..)

To properly support testing, configuration management may involve ensuring the following:

- All test items are uniquely identified, version controlled, tracked for changes, and related to each other
- All items of testware are uniquely identified, version controlled, tracked for changes, related to each other and related to versions of the test item(s) so that traceability can be maintained throughout the test process
- All identified documents and software items are referenced unambiguously in test documentation

5.5.1 Definition of Risk

Risk:

- A factor that could result in negative consequences; usually expressed as impact and like hood
- Risks are used to decide where to start testing and where to test more.
- Testing oriented towards exploring and providing information about product risks

Risk based Testing is used to reduce risk of adverse effect occurring or to reduce the impact of adverse effect

- It draws on the collective knowledge and insight of the project stakeholders to determine the risk and the level of testing required to address those risks.

5.5.2 Project Risks

A risk related to management and control of the (test) project is called as Project Risk.

Project risk include:

- **Project issues:**
  - Delays may occur in delivery, task completion, or satisfaction of exit criteria or definition of done
  - Inaccurate estimates, reallocation of funds to higher priority projects, or general costcutting across the organization may result in inadequate funding
  - Late changes may result in substantial re-work
- **Organizational issues:**
  - Skills, training, and staff may not be sufficient
  - Personnel issues may cause conflict and problems
  - Users, business staff, or subject matter experts may not be available due to conflicting business priorities

Project Risks (Cont..)

- **Political issues:**
  - Testers may not communicate their needs and/or the test results adequately
  - Developers and/or testers may fail to follow up on information found in testing and reviews (e.g., not improving development and testing practices)
  - There may be an improper attitude toward, or expectations of, testing (e.g., not appreciating the value of finding defects during testing)
- **Supplier issues:**
  - A third party may fail to deliver a necessary product or service, or go bankrupt o Contractual issues may cause problems to the project

Project Risks (Cont..)

- **Technical issues:**
  - Requirements may not be defined well enough
  - The requirements may not be met, given existing constraints
  - The test environment may not be ready on time
  - Data conversion, migration planning, and their tool support may be late
  - Weaknesses in the development process may impact the consistency or quality of project work products such as design, code, configuration, test data, and test cases
  - Poor defect management and similar problems may result in accumulated defects and other technical debt

5.5.2 Product Risks

Product Risk: it is directly related to the test object

- product risks include:
- Software might not perform its intended functions according to the specification
- Software might not perform its intended functions according to user, customer, and/or stakeholder needs
- A system architecture may not adequately support some non-functional requirement(s)
- A particular computation may be performed incorrectly in some circumstances
- A loop control structure may be coded incorrectly
- Response-times may be inadequate for a high-performance transaction processing system
- User experience (UX) feedback might not meet product expectations

5.5.3 Risk-based Testing and Product Quality

Product Risks identified during Risk-based testing are used to:

- Determine the test techniques to be employed
- Determine the extent of testing to be carried out
- Prioritize testing in an attempt to find the critical defect as early as possible
- Determine whether any non testing activities could be employed to reduce risk

Risk Management activities provide a discipline approach to :

- minimize the product failure:
- Assess and reassess what can go wrong (risks)
- Determine what risks are important to deal with
- Implement action to deal with those risks

5.6 Defect Management

Need of Defect Management Process :

- one of the objectives of testing is to find defects, defects found during testing should be logged.
- Any defects identified should be investigated and should be tracked from discovery and classification to their resolution - e.g., correction of the defects and successful confirmation testing of the solution, deferral to a subsequent release, acceptance as a permanent product limitation, etc.
- In order to manage all defects to resolution, an organization should establish a defect management process.
- This process must be agreed with all those participating in defect management, including designers, developers, testers, and product owners.

Objectives of Defect Report

- Provide developers and other parties with information about any adverse event that occurred, to enable them to identify specific effects, to isolate the problem with a minimal reproducing test, and to correct the potential defect(s), as needed or to otherwise resolve the problem

- Provide test managers a means of tracking the quality of the work product and the impact on the testing (e.g., if a lot of defects are reported, the testers will have spent a lot of time reporting them instead of running tests, and there will be more confirmation testing needed)

- Provide ideas for development and test process improvement

Summary

In this lesson, you have learnt:

- Test management is covered from a skills perspective, focusing on test execution and defect reporting and handling.
- Managing the Test Activities
- Role of test manager and testers
- Test planning activities
- Template of Test document artifacts such as test plan and test case designs
- Entry and exit criteria of tests
- Test execution activities
- Test Metrics
- Configuration Management

Review - Questions

Question 1: The degree of independence to which testing is performed is known as _____.

Question 2: there can be different test plans for different test levels. (True / False)

Question 3: Exit criteria are used to report against and to plan when to begin testing. (True / False)

Question 4: Which of the following are Test Environments

- Unit Test Environment
- QA Test Environment
- Simple Test Environment
- Product Environment

Review – Match the Following

1. Entry Criteria	A. Consultative approach
2. Exit Criteria	B. Acceptance criteria
3. Test Approach	C. Methodical approach
4. Failure based testing	D. Completion criteria

Use Cases Testing

Lesson 7: Use case Testing

Lesson Objectives

To understand the following topics:

- Use case modeling
- Advantage of use cases
- Actor
- Goals and Requirements
- Goals and scenarios
- Naming Conventions
- Alternate Path
- Exceptions
- Errors
- Precondition & Post-condition
- Good practices
- Failure scenarios

Use Case Usage

Use cases help to:

- Describe a business process
- Capture functional requirements of a system
- Document design details of a system

Use Case

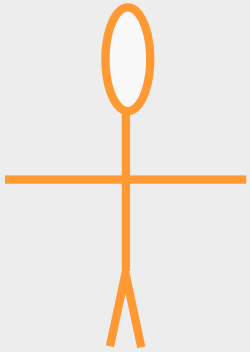
- An Use Case can be defined as a set of activities performed within a system by a User

- Each Use Case:
  - describes one logical interaction between the Actor and the system
  - defines what has changed by the interaction

Use Case Diagrams

Use Case Diagrams model the functionality of system by using Actors and Use Cases:

- Actor is a user of the system
- Use cases are services or functions provided by the system to its users



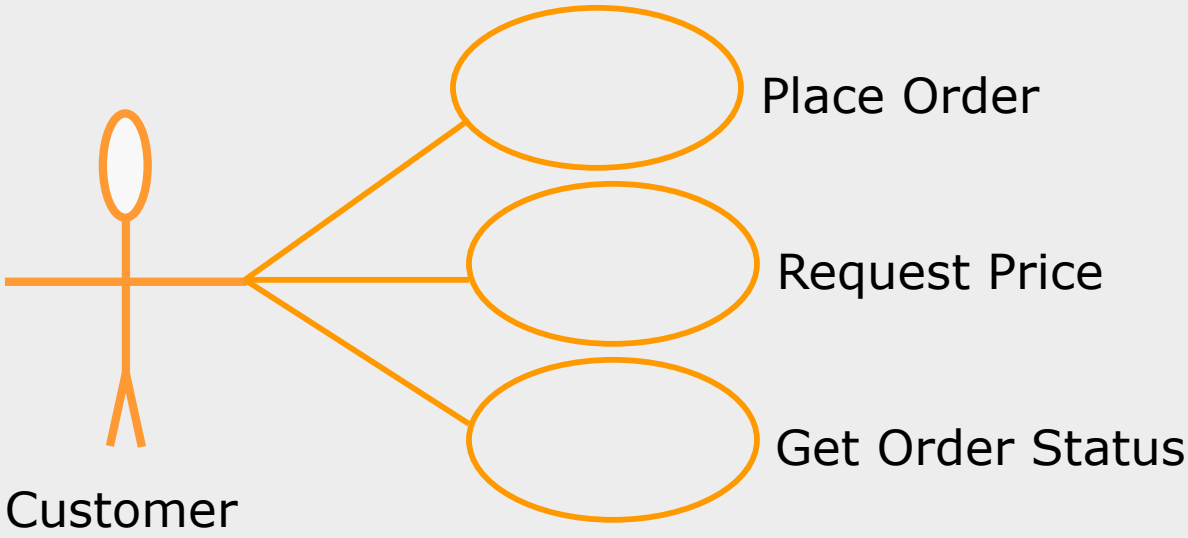
Actor

Use Case

Drawing The Use Case Diagram

A Use Case diagram has the following elements:

- Stick figure: representing an Actor
- Oval: representing a Use Case
- Association lines: representing communication between Actors and Use Cases



Advantages

- Easy to write
- Expressed in simple language that user understands
- Can be visually represented
- Can be used for validation (acceptance testing)
- Tools support
- Ideal for OO development
- Function oriented system (different types of users, complex behavioral patterns)

When Not to use case?

- System has few users
- System dominated by non-functional requirements

7.3: Actor

Definition

Actor:

- An Actor can be described as follows:
  - Actor is any entity that is external to the system and directly interacts with the system, thus deriving some benefit from the interaction
  - Actor can be a human being, a machine, or a software
  - Actor is a role that a particular user plays while interacting with the system
  - Examples of Actors are End-user (roles), External systems, and External passive objects (entities)

Type of Actors:

Primary Actor

- Initiates the Use Case
- Calls on the system to deliver a service
- Has a goal with respect to the system

Supporting Actor

- provides a service to the system under design

Use Case: Actors and goals

A GOAL is:

- What an ACTOR wants to get with the help of the system

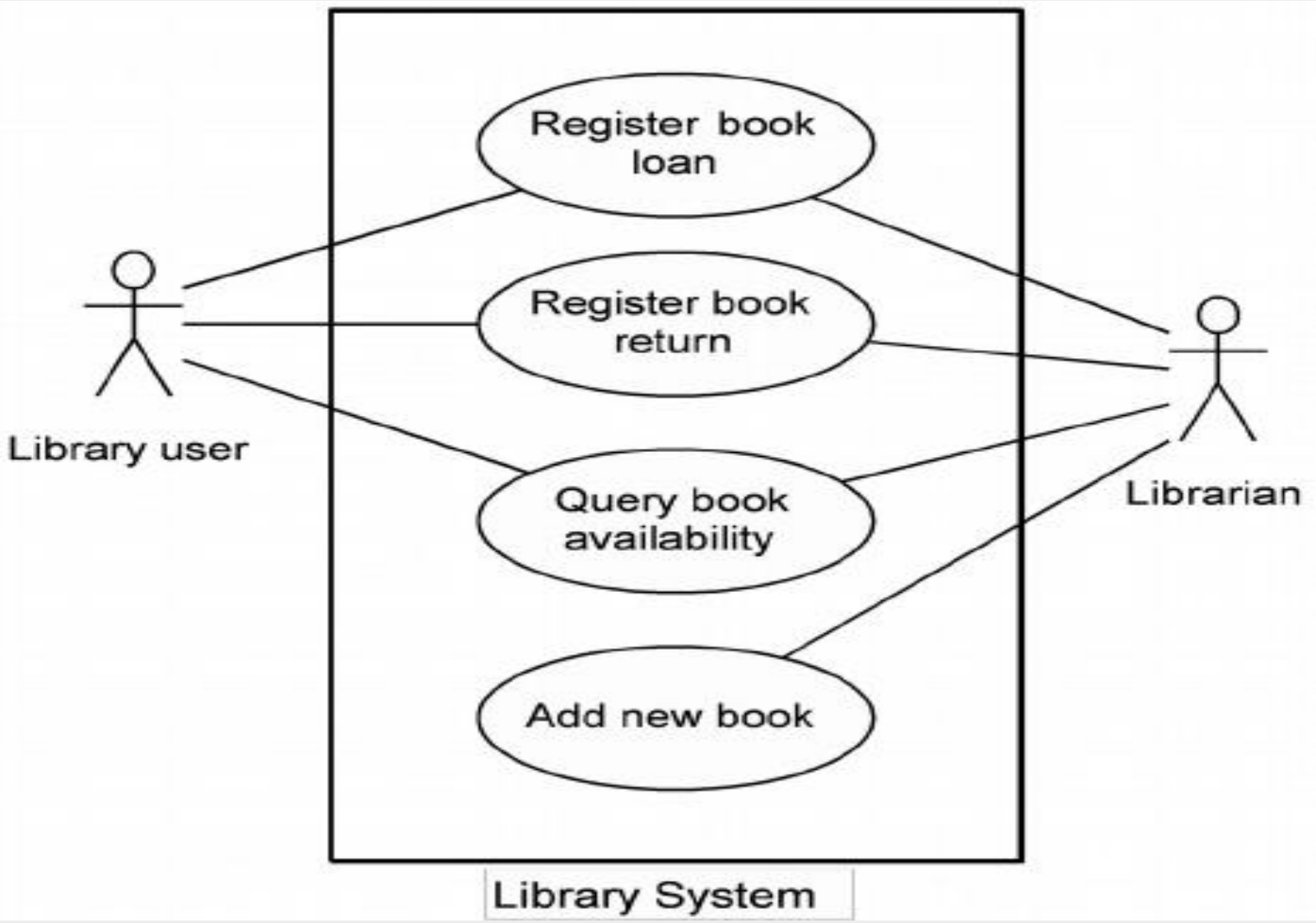
An ACTION is

- what the ACTOR must perform to reach the GOAL

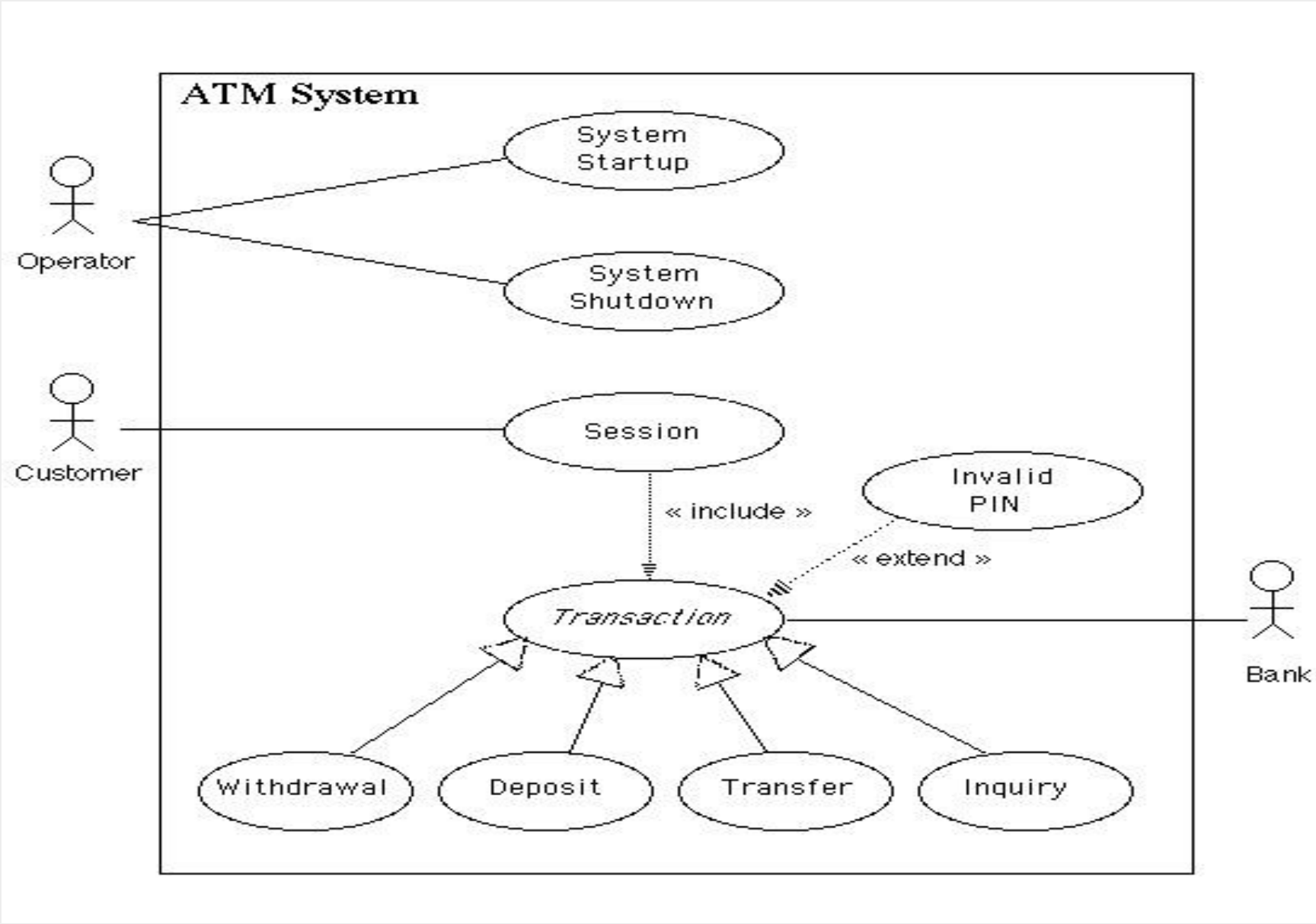
An INTERACTION is

- a sequence of steps that must be followed to complete the ACTION

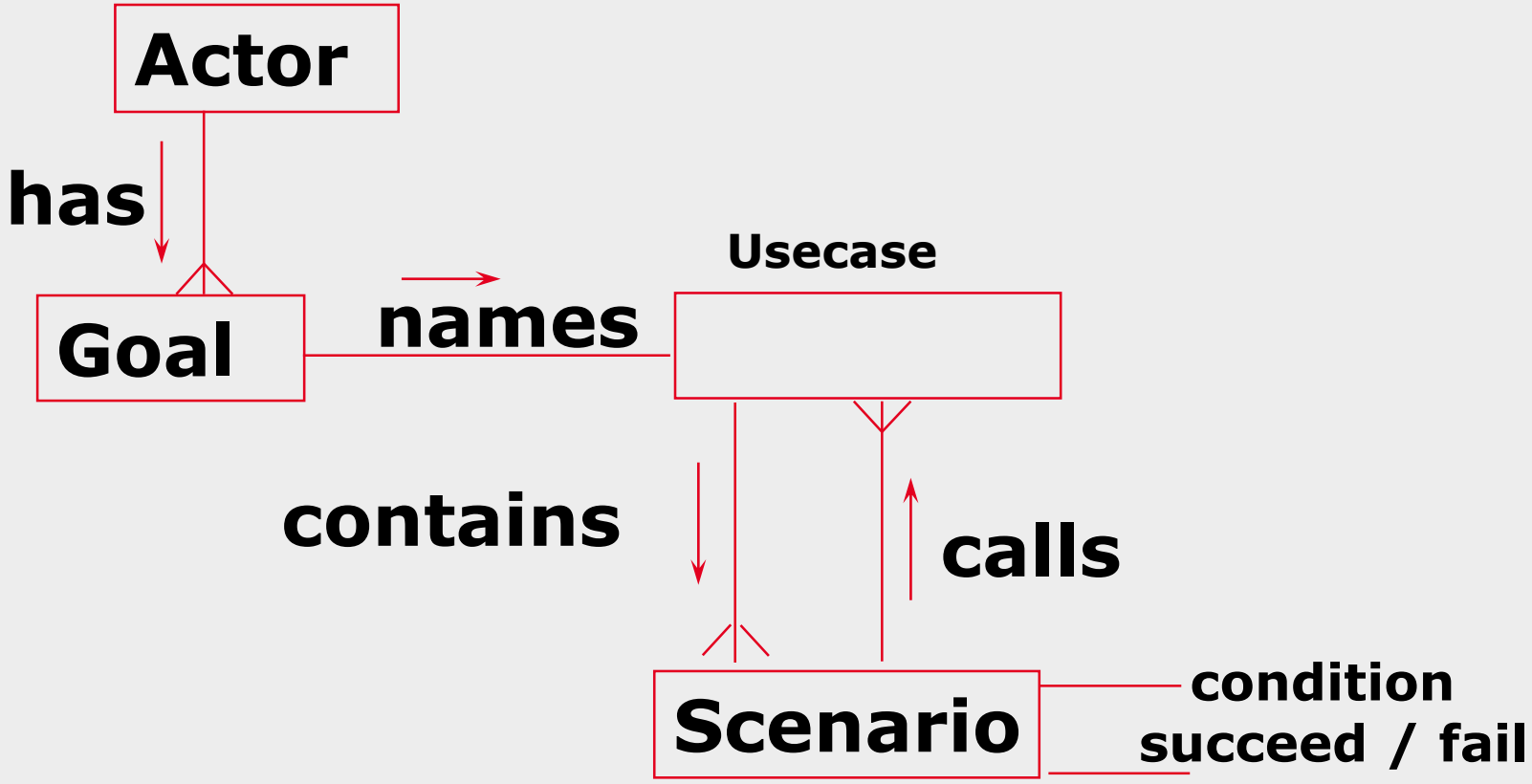
Use Case Diagram –Example 1



Use Case Diagram –Example 2



Use Cases: Actors And Goals



Use Cases: Goals And Requirements

Examining the Goals the system supports makes good functional requirements.

- “Place an order.”
- “Get money from my bank account.”
- “Get a quote.”
- “Find the most attractive alternative.”
- “Set up an advertising program.”

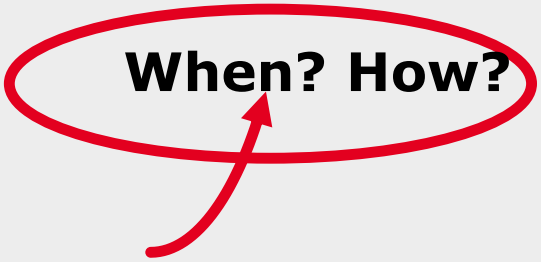
Goals summarize system function in understandable, verifiable terms of use.

Use Cases: Goals And Requirements

Structured narrative keeps the context visible

Just paragraphs:

- "ATM system has in interface with Bank database. It verifies the customer details and provides options. …"



With structured narrative:

- "Customer swipes the ATM card and enters the pin number. ATM system has an interface with Bank database to verify the customer pin. The system displays options to the customer

Goals And Scenarios

A use case pulls goals and scenarios together

In Use Case " Withdraw money from ATM"

- Scenario 1: Everything works well ...
- Scenario 2: Customer enters invalid pin …
- Scenario 3: Insufficient balance ...

Use case is goal statement plus the scenarios.

Note the grammar for Use Case: active verb first

Use Cases: Naming Actors

- Group individuals according to their common use of the system
- Identify the roles they take on when they use the system
- Each role is a potential actor
- Name each role and define its distinguishing characteristics
- Do not equate job title with role name. Roles cut across job titles
- Use the common name for an existing system; don’t invent a new name to match its role

Use Cases: Alternative Paths

For each significant action:

- Is there another significant way to accomplish it that could be taken at this point? (Variation)
- Is there something that could go wrong? (Exception)
- Is there something that could go really, really wrong (Error)

Use Cases: Exceptions

Exceptions are deviations from the typical case that happen during the normal course of events

- They should be handled, not ignored
- How to resolve them can be open to debate

What if a user mistypes her password?

What if an order can’t be fulfilled?

What if a connection to a web browser is dropped?

Use Cases: Errors

Errors are when things unexpectedly go wrong. They can result from malformed data, bad programs or logic errors, or broken hardware

- Little can be done easily to “fix things up and proceed”
- Recovery requires drastic measures

What if the disk is full?

What if equipment cannot be provisioned?

What if the OS crashes?

Use Cases: Precondition & Postcondition

Precondition

- is constraint not the event that starts the use case
- state what must always be true before beginning a scenario in use case are not tested within the use case; rather, they are conditions that are assumed to be true
- communicate noteworthy assumptions that the use case writer thinks readers should be alerted to condition to start an execution of test case

Use Cases: Precondition & Postcondition

Post conditions

- state what must be true on successful completion of the use case—either the main success scenario or some alternate path
- should be true regardless of which alternative flows were executed
- can be a powerful tool for describing use cases
- First define what the use case is supposed to achieve, the post condition

Use Cases: Good Practices

- Make the use cases clear, short and easy to read
- Use active voice, present tense, make sure actor and actor's intent are visible in each step
- Every Use case has two possible endings: Success and Failure / alternate courses. Gather both
- Create a list of primary actors and their goals (actor-goal list)
- Restrict use cases to capture system behavior….use cases are not suitable for other type of requirements

Failure Scenarios

Don’t overlook failure scenarios

- “What if their credit is too low?”
- “What if they run over the extended credit limit?”
- “What if we cannot deliver the quantity?”
- “What if data is corrupt in the database?”
- (These are commonly overlooked situations)

Summary

In this lesson, you have learnt:

- The use case model provides a complete, black-box, outside-in view of system functionality.
- A use case is a procedure by which an actor may use the system.
- A good use case is largely a sequence of interactions across the system boundary between the actor(s) and the system written in easy to understand natural language.
- An actor is always outside the system being defined
- The actor that starts the use case is the primary actor for that use case
- An actor is a role defined by the set of use cases with which it is associated

Review Question

- Question 1: _____ is very effective elicitation technique.

- Question 2: Actors live outside the boundary of the system.

  Option: True / False

- Question 3: Postconditions are basically conditions to start an execution of test case.

  Option: True / False

Review Question: Match the Following

1.Precondition	A. Non-recoverable extensions
2. Actor	B. Expectedly go wrong
3. Errors	C. These are not tested within the use case
4. Post condition	D. Unexpectedly go wrong
5. Exception	E. Must be true on successful completion of the use case
	F. Live outside the boundary of the system

Testing Concepts

Lesson 6: **Requirement Engineering**

Lesson Objectives

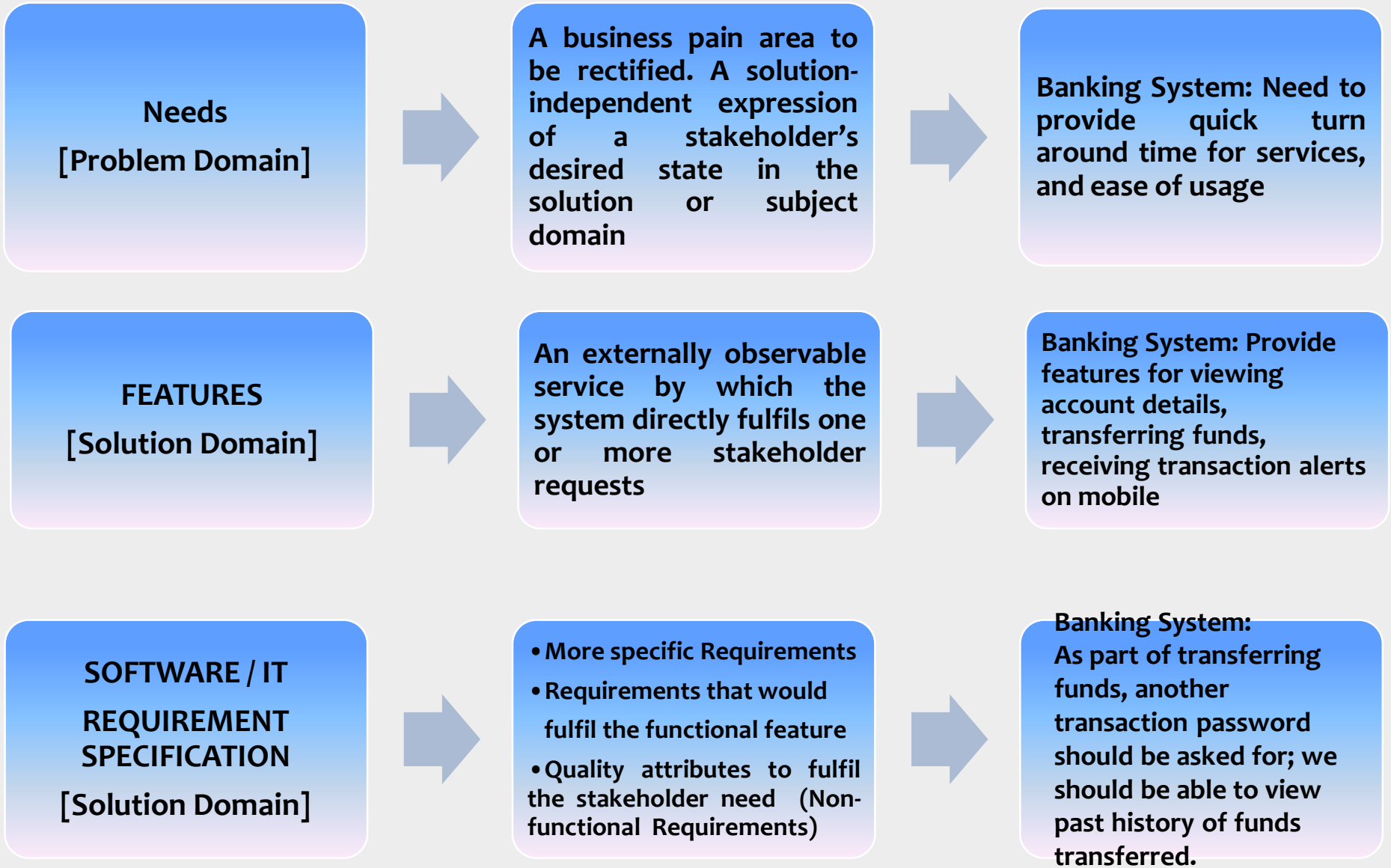
To understand the following topics:

- Evolution of Requirements
- Who provides the Requirements?
- Challenges in Requirement Gathering
- Why do we need good requirements?
  - Characteristics & Impact of bad Requirements
- Requirement engineering
- Functional Vs Non-Functional Requirements
- Non Functional Requirements: FURPS +

Lesson Objectives

- Stable and Volatile Requirements
- Baselining Requirements
- Requirements Traceability
  - Requirement Traceability Matrix
  - Maintaining Requirement Traceability
  - Requirement Traceability Matrix – Example
- Requirements Change
  - Change Management Process
  - Requirement Creep

6.1 Evolution of Requirements



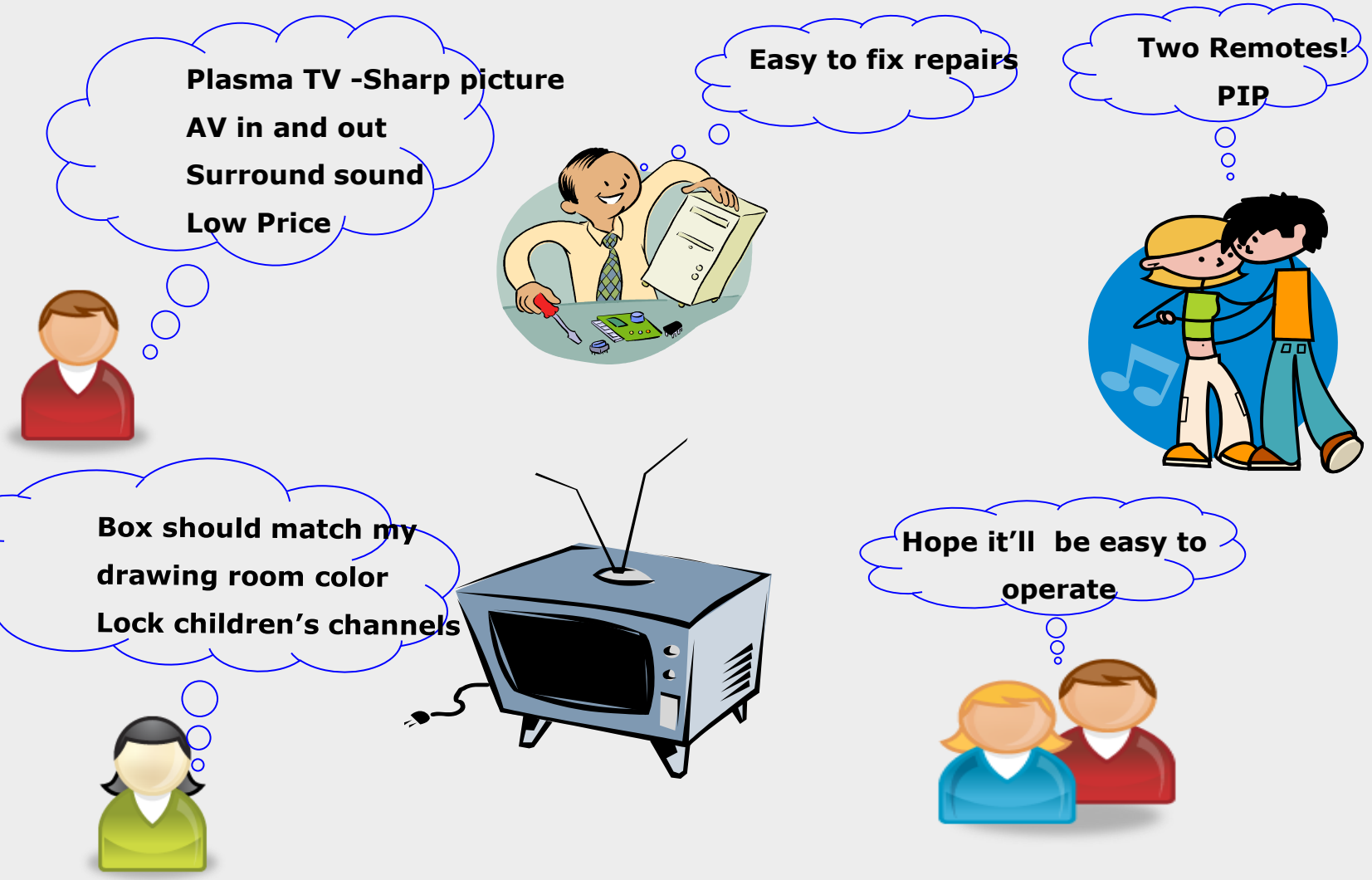
6.2 Who provides the Requirements? - Stakeholders

"Stakeholders" are the individuals who affect or are affected by the proposed software product.

Following are the different stakeholders :

- Customers – These people purchase, and/or pay for the software product in order to meet their business objectives
- End Users – use the product directly or indirectly by receiving reports, outputs, or other information generated by the product
- Development Team – They include individuals/teams from the development organization :
  - Business Analyst - elicit the requirements from the customers, users, and other stakeholders; analyze requirements, write requirements specification, and communicate the requirements to development team and other stakeholders.
  - Designers –translate the requirements into the software's architectural and detailed designs specifying how the software will be implemented.
  - Developers – implement the designs by creating the software product
  - Testers – They use the requirements for designing the test cases that they use to execute and test the software under specific, known conditions to detect the defects and provide confidence that the software performs as specified.

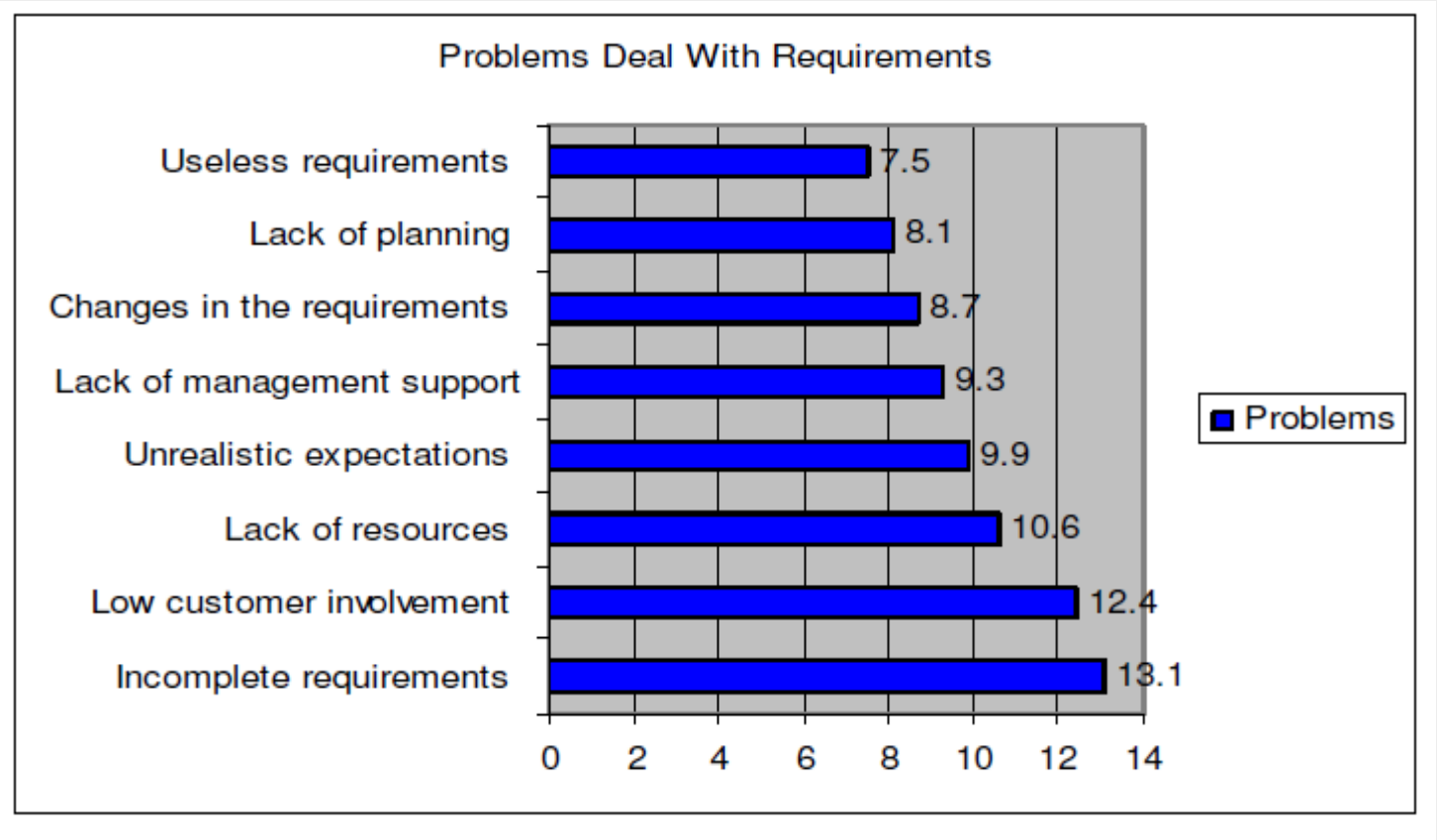
Different stakeholders….So different requirements



6.3 Challenges in Requirement Gathering

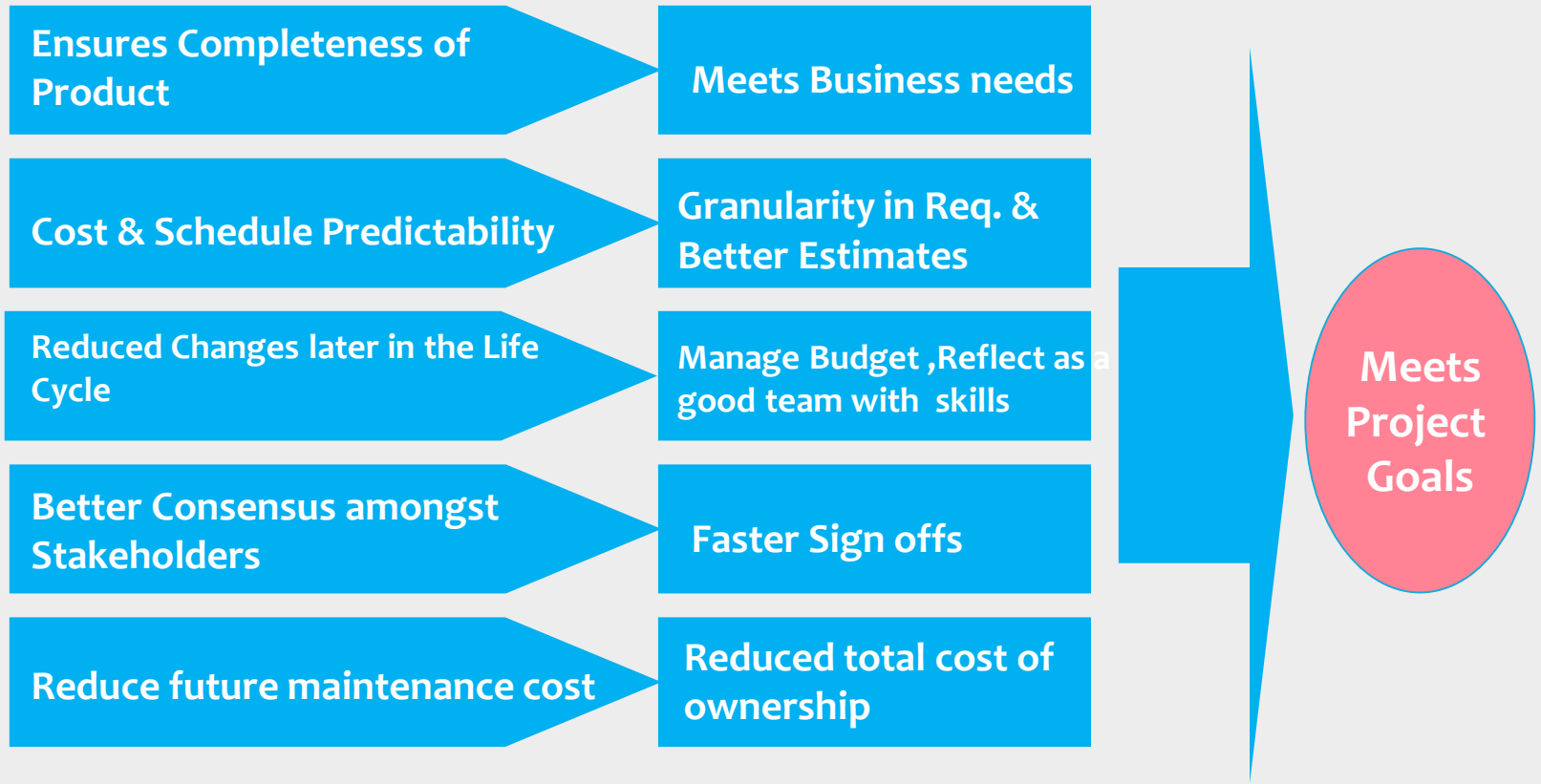
Requirement Problems are the single No.1 reason for projects to fail.

According to the Standish Group's 1995 CHAOS survey, the top two "project impaired" factors were **incomplete requirements** and **lack of user involvement.**



6.4 Why do we need good requirements?

Why do we need the requirement phase?
What is it supposed to achieve?



6.4.1 Characteristics & Impact of bad Requirements

Characteristics of defective Requirements :

- Lack of Cohesiveness
- Lack of Completeness
- Lack of Correctness
- Lack of Consistency
- Lack of Project Relevance
- Lack of Testability, Usability, Validatability
- Ambiguous

Impact of bad Requirements :

- Function failure
- Extensively over budget
- Extensively past schedule
- Extensively reduced scope
- Poor quality applications
- Not considerably used when delivered
- Sometimes getting cancelled

6.5 Requirements Engineering

Requirements Engineering is a disciplined, process-oriented approach to the definition, documentation, and maintenance of software requirements throughout the software development life cycle

Software requirements engineering is made up of two major processes: "**Requirements Development**" and "**Requirements Management**"

- Requirements development involves all of the activities that are part of eliciting, analyzing, specifying, and validating the requirements
- Requirements management involves the activities that are part of requesting changes to the baselined requirements, performing impact analysis for the requested changes, approving or disapproving those changes, and implementing the approved changes

6.6 Functional & Non-functional Requirements

Functional Requirements	Non-functional Requirements
It specifies the input and output behaviour of a systems. It defines how software behaves to meet user needs. Example : Functional requirements of a health insurance company include : • Determining Claimant Eligibility • Paying Claims • Calculating Premium	It represents quality attributes of the system : Availability, Maintainability, Performance, Portability, Reliability, Robustness, Security, Scalability etc. Examples: • It should be easy to see the history of transactions • The system should be available 99.9% of the time • The system should use SSH public-key cryptography to authenticate the remote computer and allow the remote computer to authenticate the user, if necessary • The system should be able to serve at the most 100 concurrent users

Functional Vs Non-Functional Requirements

6.7 Non Functional Requirements: FURPS +

Functionality
Usability
Reliability
Performance
Supportability
+ other such quality attributes

6.8 Stable and Volatile Requirements

Stable Requirements

- They are related to the core activities of the system and its domain
- For example, in an organization there will be requirements concerned with employees, departments, payroll etc.

Volatile Requirements

- These are requirements that are likely to change during the system development process or after the system has been become operational
- Examples of volatile requirements are requirements resulting from organization’s leave policies or Income Tax policies enforced by the country’s government bodies

6.9 Baselining Requirements

The requirements are baselined at the end of the Requirements Development phase & ideally signed-off by the customer

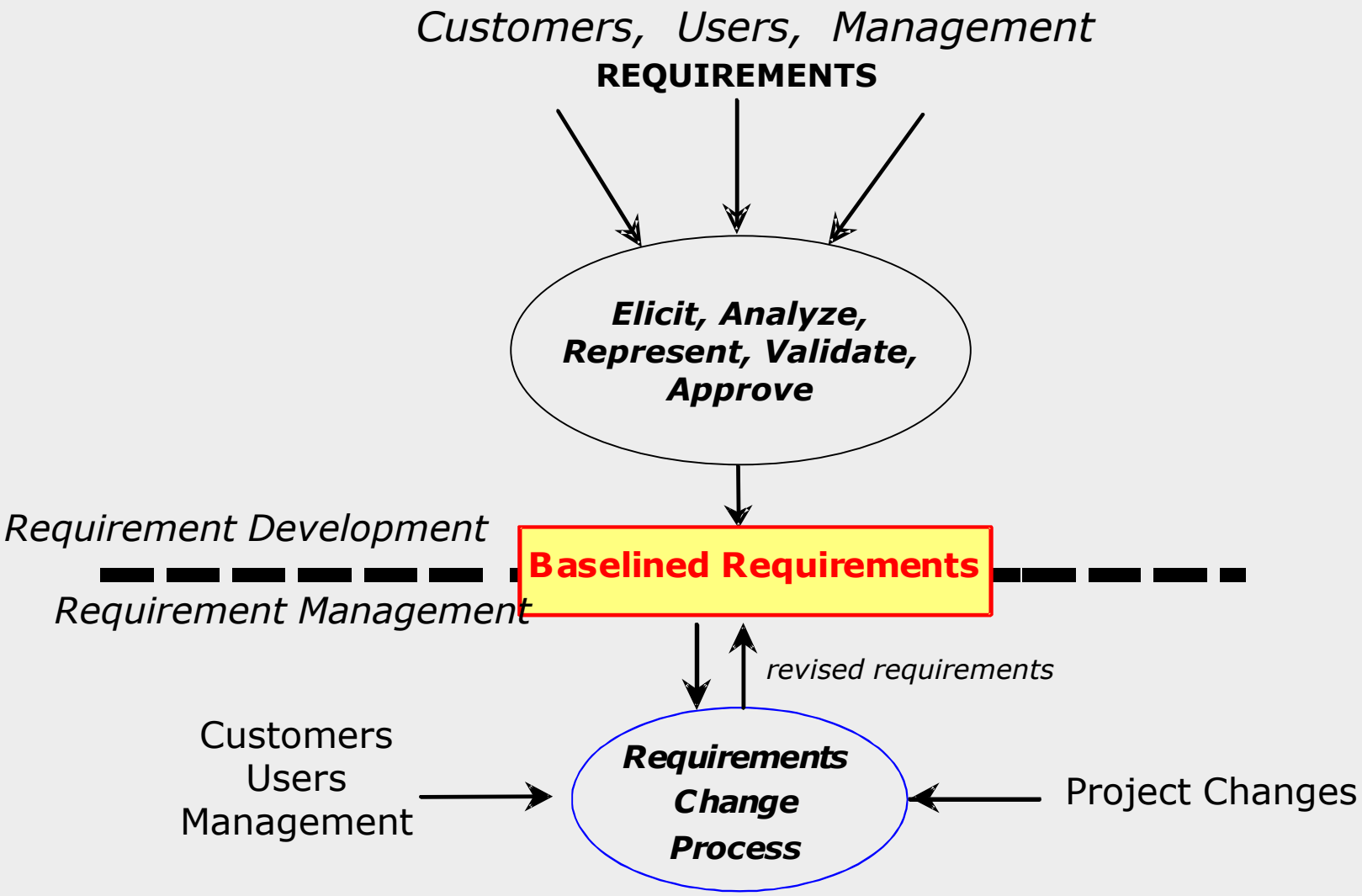
A requirements baseline is:

- A snapshot in time of a set of requirements
- Used as a mechanism to track changes as the project progresses
- Constitutes agreement on scope between customer and development team

Scope drives estimate, schedule, staffing, deadlines

New baselines are typically created at major project milestones

Baselining Requirements (Cont..)



6.10 Requirements Traceability

It is one of the essential activities of good requirements management

Requirement traceability helps in assessing the impact of requirements change

Traceability is used to track the relationship between each unique product-level requirement and its source

> ***"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another."***
>
> ***The IEEE Definition***

6.11.1 Requirement Traceability Matrix

RTM is a table containing requirements of a project and their relation to the engineering work products

It ensures completeness in translating requirements to the delivered work products

Advantages

- Ensures completeness of testing against requirements
- Facilitates the impact analysis of the requirements change on all the related work products
- Enables scope analysis for regression testing
- Helps judging requirements stability from a customer
- Helps to analyze Requirements creep

Suggested Tools

- Rational Requisite Pro
- Excel Sheet (Traceability template)

6.11.2 Requirement Traceability Matrix – Example

Example : XYZ Banking Application

This is a site that aims at computerizing the business process of XYZ bank. The following are the business targets that are to be achieved :

Step 1: Business Requirement Document (BRD) :

BR ID	Module Name	Roles Involved	Description
BR_1	Login	Customer Manager	A customer can login using login module. A Manager can login using login module.
BR_2	Enquiry	Customer Manager	A customer can view balance of his accounts only. A manager can view balance of all customers who come under his supervision.
BR_3	Fund Transfer	Customer Manager	A customer can transfer funds from his 'own' account to any destination account. A manager can transfer funds from any account under his supervision.
BR_4	Loan Process	Customer Manager	A customer can access loan information and apply for loan. A manager can grant, reject, suggest changes to the loan request under his supervision.

Requirement Traceability Matrix – Example

Below is our Technical Requirement Document (TRD) based on the interpretation of the Business Requirements Document (BRD)

Step 2: Technical Requirement Document (TRD) for **BR_1 (Login)**

TR ID	Module Name	Description
TR_01	Login Module	User ID and Pwd must not be blank
TR_02	Login Module	If User ID and password are valid. Login.
TR_03	Login Module	The Pwd field should not allow copy paste from any source.

Requirement Traceability Matrix – Example

Step 3: On the basis of Business Requirement Document (BRD) and Technical Requirement Document (TRD), testers start writing test scenarios.

Test Scenarios :

Test Scenario ID	Test Scenarios
TS_1	Verify with valid User Credentials
TS_2	Verify with invalid User Credentials
TS_3	Verify the length provided for the user name field
TS_4	Verify the length provided for the Pwd field
TS_5	Verify copying the Pwd from external file and pasting it into Pwd field.

Requirement Traceability Matrix – Example

Step 4: For each test scenario, write at least 1 or more test cases.

Test Cases :

Test Case ID	Test Condition	Test Steps	Test Data	Expected Result
TC_1	To validate that user is able to login successfully with valid User ID & valid Pwd.	1.Go to Login Page 2.Enter User ID 3.Enter Password 4. Click Login	user1 1234B	Login Successful
TC_2	To validate that user is unable to login with invalid User ID & valid Pwd.	1.Go to Login Page 2.Enter User ID 3.Enter Password 4. Click Login	sampleuser 1234B	Login Failure
TC_3	To validate that user is unable to login with valid User ID & invalid Pwd.	1.Go to Login Page 2.Enter User ID 3.Enter Password 4. Click Login	user1 samplePwd	Login Failure

Requirement Traceability Matrix – Example

Step 5: You can now start creating RTM.

Identify the Test scenarios, Technical Requirements and Business Requirements that the test cases are verifying.

BR ID	TR ID	TS ID	TC ID
BR_1	TR_01	TS_3	
		TS_4	
	TR_02	TS_1	TC_1
		TS_2	TC_2
		TS_2	TC_3
	TR_03	TS_5	
BR_2			
BR_3			
BR_4			

At this stage, the RTM can be used to find gaps. For example, in the above RTM, you see that there are no test cases written for TS_3, TS_4, TS_5.

it will indicate the places where the test team needs to work some more to ensure 100% coverage.

Requirement Traceability Matrix – Example

Step 6: Expand the RTM to include test case execution status and defects.

BR ID	TR ID	TS ID	TC ID	Status	Defect ID
BR_1	TR_01	TS_3	TC_5	Pass	
		TS_3	TC_6	Fail	D_05
		TS_4	TC_7	Pass	
	TR_02	TS_1	TC_1	Pass	
		TS_2	TC_2	Fail	D_11
		TS_2	TC_3	Pass	
	TR_03	TS_5	TC_10	Fail	D_02

6.11 Requirements Change

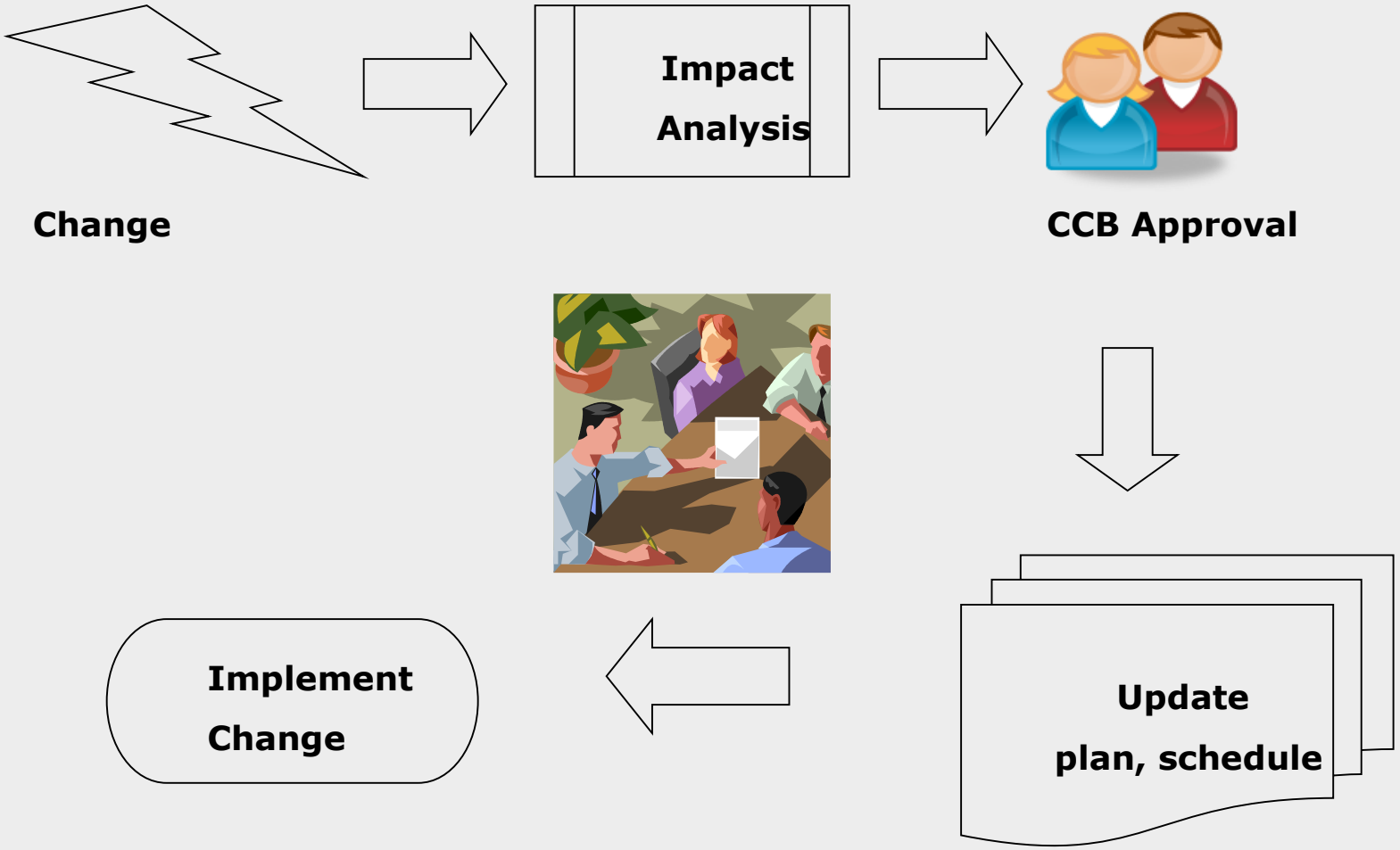
Users **can** propose **requirements change** at any stage of **SDLC.**

Requirements change because :

- Understanding of the problem improved : Customer's expectations change once they see the product taking shape
- Initial elicitation activities are imperfect : We failed to ask the right people the right questions at the right time.
- The priority of requirements from different viewpoints change during the development process
- Business needs evolve
- The users changed their perceptions : Customers may specify requirements from a business perspective that conflict with end-user requirements
- The external environment changed : The business and technical environment of the system changes during its development
- We either failed to create or follow a process to help manage change.

6.11.1 Change Management Process

Establish a formal process for managing changes to the system requirements



Change Management Process

Identify potential change

- Require new functionality
- Encounter problem
- Request change

Do functional impact assessment upon any change request

Analyze change request

- Determine technical feasibility
- Determine costs and benefits

Evaluate change

Obtain approval from customer on scope of change, impact & efforts needed

Plan change

- Analyze change impact
- Create planning

Implement change

- Execute change
- Propagate change
- Test change
- Update requirement artifacts with new requirement
- Release change

Review and close change

- Verify change
- Close change

6.11.2 Requirement Creep

> ***"Scope creep (also called requirement creep and feature creep) in project management refers to uncontrolled changes or continuous growth in a project's scope. This can occur when the scope of a project is not properly defined, documented, or controlled."***

Why does Requirement Creep occur ?

- Poor requirement analysis
- Not involving customers early though
- Insufficient detailing on the complexity of the project
- Lack of change control
- Gold Plating
- Unwillingness to say no to a client

Requirement Creep (Cont..)

- Even when there is a clearly defined project scope, one must be aware that Requirement creep can still occur during project development.
- Requirement creep tends to additional requirements needed to achieve the new objectives. This can overwhelm the capacity of the resources allocated to the project resulting into project missing deadlines, budgets or complete failure.
- Therefore, preventing requirement creep and managing requirement creep is the key to successful project management.

Measures to control Requirement Creep

Following are some of the common measures those can be used to minimize requirement creep

- Utilize various techniques for more thoroughly defining user requirements up front
- Involve the customers in the earliest stages of the project possible
- Achievable goals should be set
- Prioritize requirements into must-haves versus nice-to-haves
- Project managers have to learn when to say no and when to say yes
- When the client wants to change or add a requirement, the change or addition should be analyzed for resource, cost, and schedule impacts
- Perform constant internal review to make sure the project is on track and within scope
- Set a timeline or due date for all tasks
- Have a tracking system for tasks, due dates, and action items

Review Question

Question 1: Which of the non functional requirements can ensure that the application should be available for German & Japanese users?

Question 2: In which of the requirement gathering patterns, requirements are specified in detail and passes thru multiple reviews and sign-offs?

Question 3: Volatile Requirements are likely to change during the system development process or after the system has been become operational. (T/F)

Question 4: The requirements are baselined at the beginning of the Requirements Development phase & ideally signed-off by the development team. (T/F)

Question 5: Which type of requirement traceability is used to validate whether the project is evolving in the desired direction and for the right product?