Software Testing

Unit 2

Unit Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

PES UNIVERSITY

List of Contents

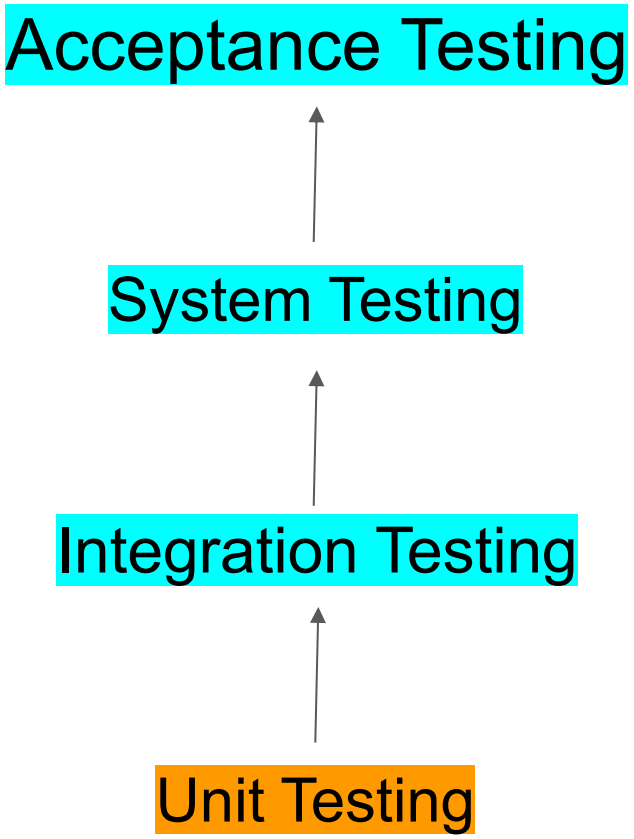
Unit Testing – Introduction

- Individual units or components of a software are tested.
- Purpose is to validate that each unit of the software code performs as expected.
- Done during the development (coding phase) of an application by the developers.
- A unit may be an individual function, method, procedure, module, or object.
- Unit testing is first level of testing done before integration testing.
- White Box testing technique that is usually performed by the developer.

Why Unit Testing

1. It reduces defect cost.
2. Saves time and money.
3. Increases confidence in code maintenance.
4. Makes code reusable.
5. Increases development speed.
6. Makes code more reliable.
7. Makes debugging more easy.

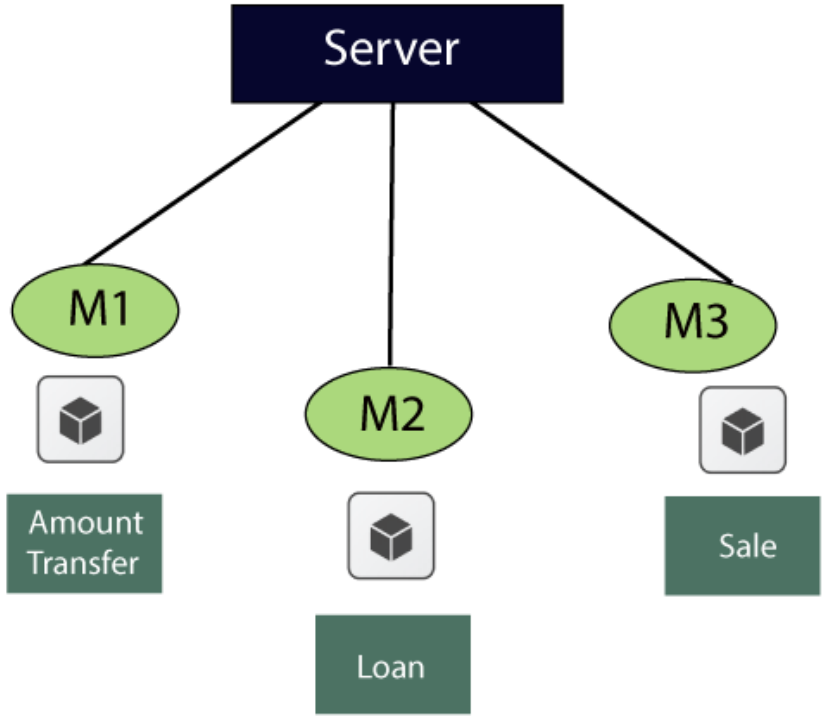
Unit Testing – In the Testing Cycle



1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the testing code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code reuse. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.

Unit Testing - Example

Let us see one sample example for a better understanding of the concept of unit testing:



Follow this link to view detailed example referring to the above image : https://www.javatpoint.com/unit-testing

How to do Unit Testing? - Modes

Unit Testing is done in two modes:

- Manual
- Automated

Unit testing is commonly automated but may still be performed manually. Software Engineering does not favor one over the other but automation is preferred.

Unit Testing - How it works?

Under the automated approach-

- A developer writes a section of code in the application just to test the function. They would later comment out and finally remove the test code when the application is deployed.
- A developer could also isolate the function to test it more rigorously. This is a more thorough unit testing practice that involves copy and paste of code to its own testing environment than its natural environment. **Isolating the code helps in revealing unnecessary dependencies between the code being tested and other units or data spaces in the product**. These dependencies can then be eliminated.
- A coder generally uses a UnitTest Framework to develop automated test cases. Using an automation framework, the developer codes criteria into the test to verify the correctness of the code. During execution of the test cases, the framework logs failing test cases. Many frameworks will also automatically flag and report, in summary, these failed test cases. Depending on the severity of a failure, the framework may halt subsequent testing.
- The workflow of Unit Testing is :
    1. Create Test Cases
    2) Review/Rework
    3) Baseline
    4) Execute Test Cases.

Unit Testing – Advantages and Dis-advantages

1. Advantages

2. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API.
3. Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. Regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.
4. Due to the modular nature of the unit testing, we can test parts of the project without waiting for others to be completed.

1. Dis-advantages

2. Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs
3. Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

Testing Methodologies

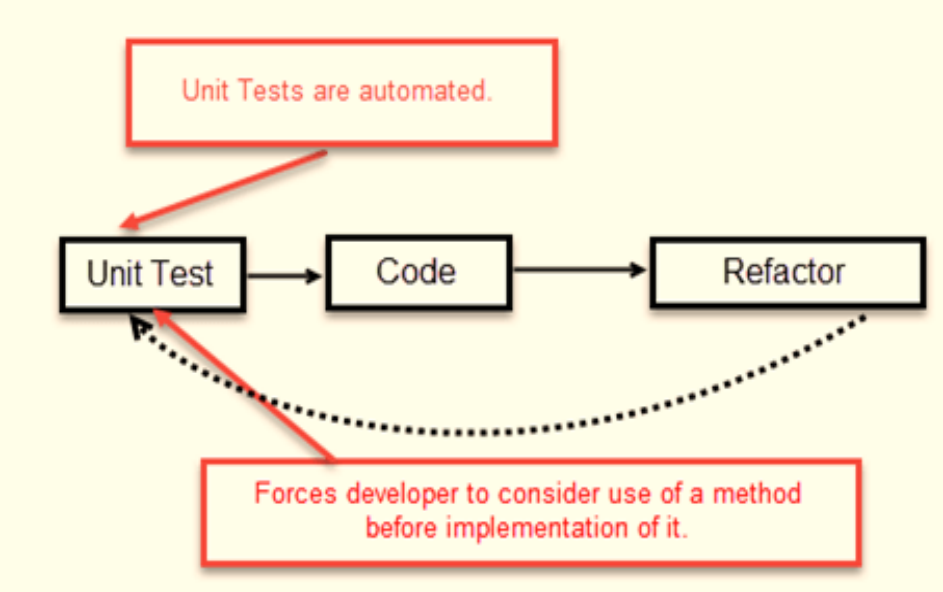
Categorized into three parts

1. Black box testing - that involves testing of user interface along with input and output
2. White box testing - that involves testing the functional behaviour of the software application
3. Gray box testing - that is used to execute test suites, test methods, test cases and performing risk analysis.

Test Driven Development - TDD

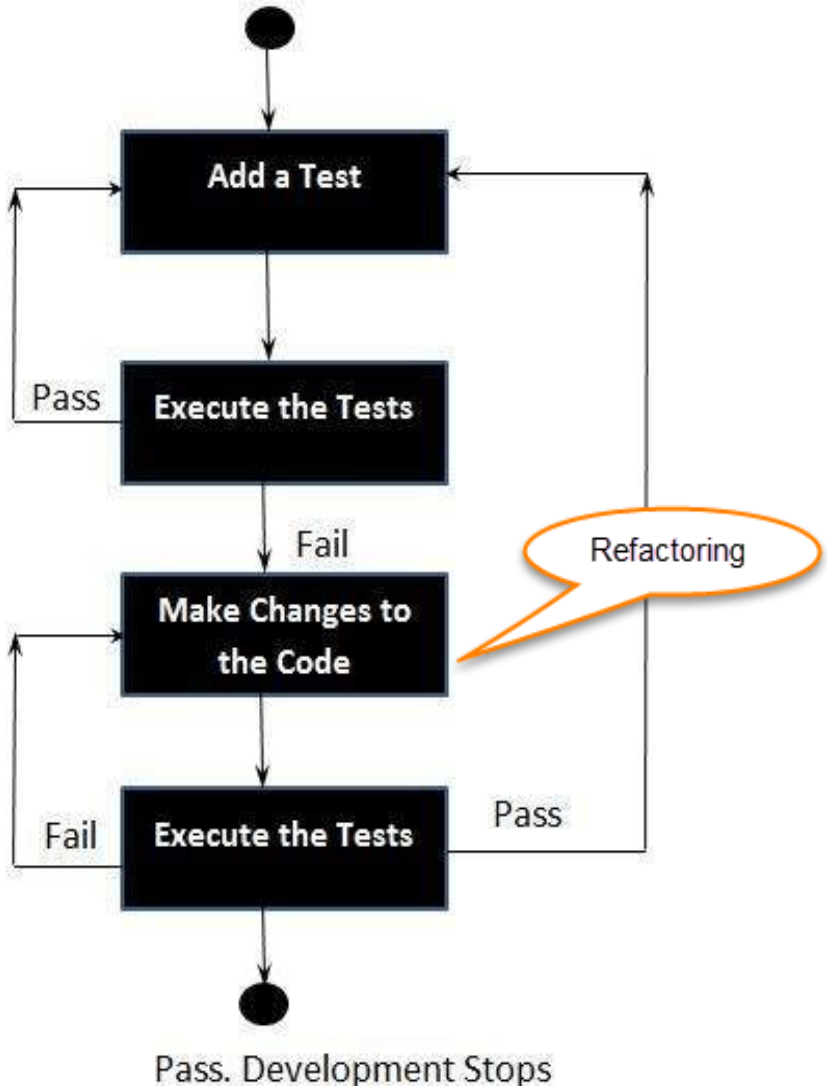
1. A new testing technique by advocates of Extreme Programming.
2. *Introduction to TDD : "Test-Driven Development" by Kent Beck.*
3. Two simple rules
   a. Write a failing automated test before writing any code
   b. Remove duplicates

TDD process



Test-Driven Development (TDD) involves designing and creating tests for each application function, with developers writing new code only when automated tests fail. This prevents code duplication and encourages writing minimal code to pass tests, prioritizing requirement conditions.

Steps in TDD



Following steps define how to perform TDD test,

1. Add a test.
2. Run all tests and see if any new test fails.
3. Write some code.
4. Run tests and Refactor code.
5. Repeat.

TDD Advantages

1. It promotes affirmative testing of the application and it's specifications.
2. TDD makes code simpler and clear.
3. Reduces the documentation process at developers end.

Refactoring

- Martin Fowler, the pioneer of refactoring, compiled industry best practices into a defined list of refactorings and detailed their implementation in his book "Refactoring: Improving Existing Code."

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal architecture.

- Refactoring can help mitigate code thrashing, a situation where different developers introduce conflicting changes. By following established refactoring patterns and principles, teams can minimize conflicts and improve code collaboration.

- Refactoring improves code by making it:
  - More efficient by addressing dependencies and complexities.
  - More maintainable or reusable by increasing efficiency and readability.
  - Cleaner so it is easier to read and understand.
  - Easier for software developers to find and fix bugs or vulnerabilities in the code.

Unit Testing Checklist

1. Write unit tests for all code
1. Don’t postpone the tests
1. Ensure that the code passes unit tests and integration tests before check-ins
1. Use automated tools such as NUnit
1. Check the tests and any other setup files into SCC
1. Consider using TDD for 100% coverage
1. Refactor code only when there are thorough set of tests for the code

Unit Testing Tools

There are several automated unit test softwares available:

1. **Junit:** Junit is a free to use testing tool (i.e. Java unit testing framework) used for Java programming language. It provides assertions to identify test method. This tool test data first and then inserted in the piece of code.
2. **NUnit:** NUnit is widely used unit-testing framework use for all .net languages. It is an open source tool which allows writing scripts manually. It supports data-driven tests which can run in parallel.
3. **JMockit:** JMockit is open source Unit testing tool. It is a code coverage tool with line and path metrics. It allows mocking API with recording and verification syntax. This tool offers Line coverage, Path Coverage, and Data Coverage.

Assert is a method useful in determining Pass or Fail status of a test case.
Assertion methods are useful in writing test cases and to detect test failure.
The assert methods are provided by the class org.junit.Assert which extends java.lang.Object class.
Types of assertions : Boolean, Null, Identical etc.
Ref: https://www.guru99.com/junit-assert.html

Unit Testing Tools (Cont.)

There are several automated unit test softwares available:

1. **EMMA**: EMMA is an open-source toolkit for analyzing and reporg code written in Java language. Emma support coverage types like method, line, basic block. It is Java-based so it is without external library dependencies and can access the source code.
2. **PHPUnit**: PHPUnit is a unit testing tool for PHP programmer. It takes small portions of code which is called units and test each of them separately. The tool also allows developers to use pre-define assertion methods to assert that a system behave in a certain manner.

THANK YOU

Introduction to Software Quality & Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

Software Testing

Unit 2

White Box Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

List of Contents

What is White Box Testing

- White Box Testing is the testing based on the knowledge of how the software solution's internal structure is.
- It focuses primarily on strengthening quality of implementation proving design and usability.
- White box testing is also known as **clear box, open, structural, and glass box testing**.
- Requires access to code
- Looks at the program code and performs testing by mapping program code to functionality

“Testing can prove presence of defects and not their absence!”

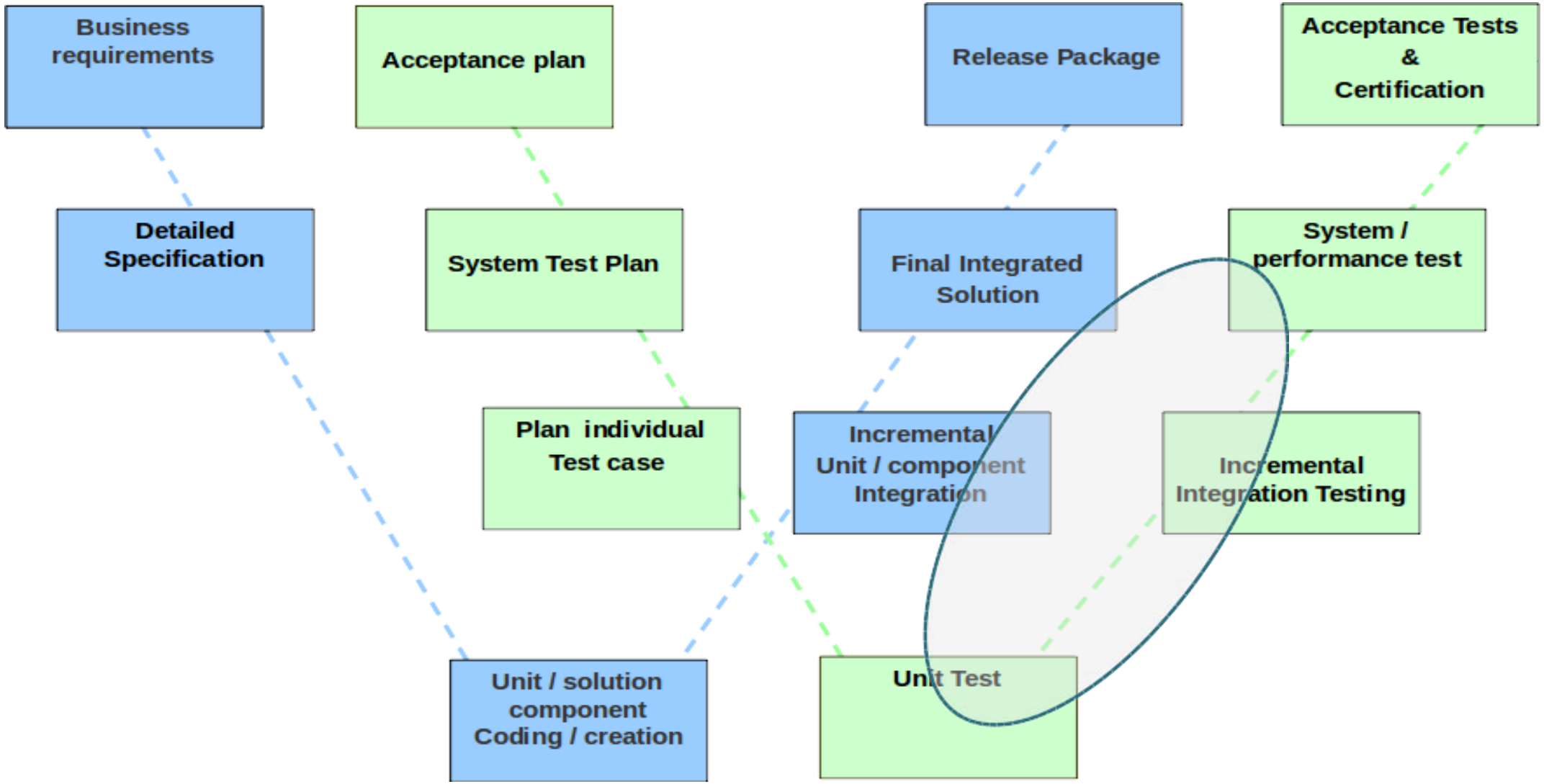
Objectives of White Box Testing

- Finding Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- The functionality of conditional loops
- Internal security holes
- Defects due to common programming mistakes
- Testing of each statement, object and function on an individual basis
- Identify boundary values based on implementation structure
  - Buffer sizes for example
- Identify violations to “construction” standards
  - Coding standards
  - Design specified guidelines
  - Professional standards

White Box Testing is needed

- Early defect identification
- Early confidence building
- Reduce complexity in integrated/blackbox testing
- Address both functionality and quality aspects
  - Maintainability, reliability, availability

White Box Testing in SDLC



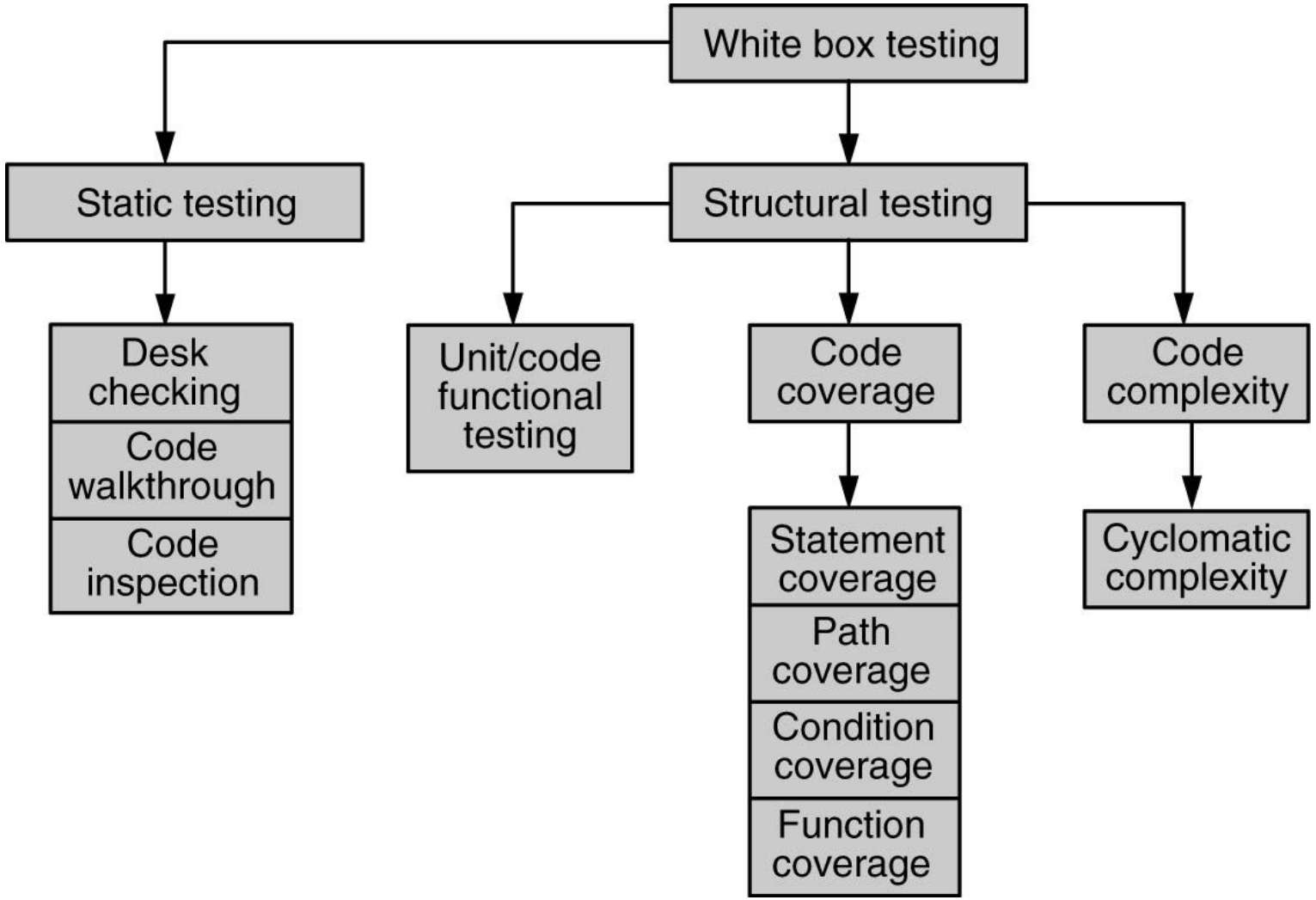
Why White Box Testing

- The program code truly represents what the program actually does and not just what it is intended to do!
- It minimizes delay between defect injection and defect detection (i.e., does not postpone detection to be done by someone else).
- Can catch “obvious” programming errors that do not necessarily map to common user scenarios (e.g., divide by zero).

Limitation of White Box Testing

- Tendency to miss the big picture
- Implementation technology skill required
  - Progg. Language, database, environment, runtime behavior
  - Deeper knowledge required
  - Including dev skills & IDE
- Effort is significantly more
  - Additional development may be required
- Source / documentation may not be available – fully / partly.
- Developers might develop blind spots for their own defects
- Fully covered code may not correspond to realistic scenarios

Types of White Box Testing



Quote

“The human eye has an almost infinite capacity for not seeing what it does not want to see. Programmers, if left to their own ways, will ignore/overlook the most glaring errors in their output—errors that anyone else can see in an instant."

***- The Psychology of Computer Programming*, Gerald Weinberg**

Static Testing

- Involves only the source code and not the executable
  - Does not involve executing the program on a machine but rather humans going through source code it or the usage of specialized tools [No computer required!]
- Key aspects that are looked into
  - Whether the code works according to the functional requirement
  - Whether the code has been written in accordance with the design developed earlier in the project life cycle
  - Whether the code follows all applicable standards
  - Whether the code for any functionality has been missed out
  - Whether the code handles errors/exceptions properly
  - Whether the code follows professional / best practices
    - What are these ?
    - Obvious arithmetic / logical mistakes

Static Testing by Humans

- Humans can find errors that computers can’t.
- Multiple humans can provide multiple perspectives.
- Errors due to ambiguities can be realized
- A human evaluation can compare the code against the specifications more thoroughly.
- It can detect multiple defects at one go - Helps root cause identification
- It reduces downstream, inline pressure.
- Verify difficult to execute paths during testing special & rare conditions.

- But, Who does the testing?
  - Depends on type of static testing – see next.

Static Testing types

- Different types [All are called ‘Code Review’]
  - Desk checking of the code
  - Code walkthrough
  - Code inspection

- “QA vs QC” argument
  - Increasing involvement of people
  - More variety of perspectives
  - Increasing formalism
  - Increasing the likelihood of identifying complex defects

Desk Checking (Personal reviews)

- Author informally checks the code against the specifications and corrects defects found.
- No structured method or formalism is required to ensure completeness.
- No log or checklist is maintained.
- It relies only on the thoroughness of the author.
- It may suffice for “obvious” programming errors but may not be effective for incomplete / misunderstood requirements.

Advantages and Disadvantages of Desk Checking

- Advantages
  - The programmer knows the code and programming language well and hence is best suited to read the program
  - Less scheduling and logistics overheads
  - Reduces delay in defect detection and correction
- Disadvantages
  - Person dependent, not scalable or reproducible
  - Tunnel vision of developers
  - Developers prefer writing new code and don’t like testing

Code Walkthrough

- Group oriented (as against desk checking)
- Brings in multiple perspectives
- Multiple specific roles (author, moderator, inspector, etc.), as discussed in Fagan Inspection
- Simpler Variant
  - Another peer/Senior person goes through the code alone after an initial brief and occasional clarifications
  - More common and efficient too.

Code Inspection / Fagan Inspection

- Group-oriented activity
- Highly formal and structured
- Multiple participants with specific roles
- Requires thorough preparation
- Documentation of diverse views
- Review in a defined structured manner.

Roles in a Formal Inspection

- **Author**
  - Author of the work product
  - Makes available the required material to the reviewers
  - Fixes defects that are reported

- **Moderator**
  - Controls the meeting(s)

- **Inspectors** (reviewers)
  - Prepare by reading the required documents
  - Take part in the meeting(s) and report defects

- **Scribe**
  - Takes down notes during the meeting
  - Assigned in advance by turns
  - Can participate to review to the extent possible
  - Documents the minutes and circulates them to participants

Documents in a Fagan Inspection

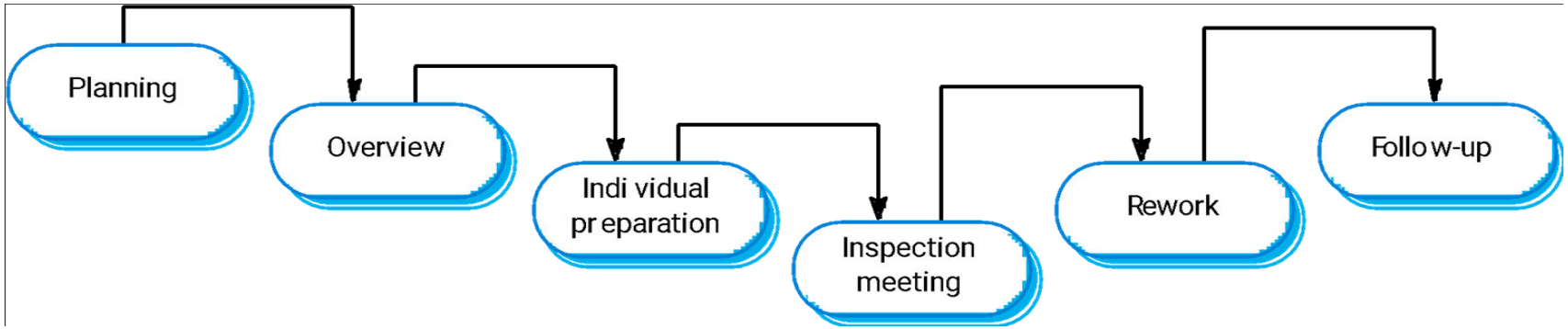
Typical documents circulated:

- Program code
- Design / program specifications
- SRS (if needed)
- Any applicable standards (e.g., coding standards)
- Any necessary checklists (e.g., code review checklist)

Meetings in a Fagan Inspection

- **Preliminary meeting (optional)**
  - Author explains his / her perspective
  - Makes available the necessary documents
  - Highlights concern areas, if any, for which review comments are sought
- **Defect Logging Meeting**
  - All come prepared!
  - Moderator goes through the code sequentially
  - Each reviewer comes up with comments
  - Comments / defects categorized as “defect” / “observation,” “major” / “minor,” “systemic” / “mis-execution”
  - Scribe documents all the findings and circulates them
- **Follow-up meeting (optional)**
  - After Author fixes defects
  - If required, a follow-up meeting is called to verify completeness of fixes

Inspection Process



Advantages and Disadvantages of Fagan Inspection

- Advantages
  - Thorough, when prepared well
  - Brings in multiple perspectives
  - Has been found to be very effective

- Disadvantages
  - Logistically difficult
  - Time consuming
  - May not be possible to exhaustively go through the entire code

Tool Usage for Static Analysis

- Whether there are unreachable codes
- Variables declared but not used
- Mismatch in definition and assignment of values to variables
- Illegal or error-prone type-casting of variables
- Use of non-portable or architecture-dependent programming constructs
- Memory allocated but not having corresponding statements for freeing up memory
- For calculation of **cyclomatic complexity**
- As an extension of compilers (lint, compiler flag driven checking…)
- Coding standards adherence
- Variable naming, indentation, in-line documentation……
- Best practice violations – Hard codings…

https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/

Static Analysis Tool - LINT

```
#include <stdio.h>
printarray (Anarray)
int Anarray;
{   printf("%d",Anarray);  }

main ()
{
 int Anarray[5]; int i; char c;
 printarray (Anarray, i, c);
 printarray (Anarray) ;
}
```

```
---------------------- Tool execution
lint lint_ex.c
 ---------------------- Output
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

THANK YOU

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

Software Testing

Unit 2

Structural Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

List of Contents

Structural Testing

- Done by running the executable on the machine
- Entails running the product against some predefined test cases and comparing the results against the expected results
- Designing test cases and test data (even test programs – stubs)to exercise certain portions of code
- Types of structural testing
  - Unit / code functional testing
  - Code coverage testing
  - Code complexity testing

Unit/Code Functional Testing

- Initial quick checks by developer
- Removes “obvious” errors
- Done before more expensive checks
- Building “debug versions”
- Running through an IDE

- “Debugging” or “testing”? (Who cares?!)

- Done at programmer level
- Most of the times – “indirect & controlled” execution because other parts of programs not ready
- Stubs, test harnesses….
- Unit testing tools – Junit…

Code Coverage Testing

- Focus is to cover all / required parts of code
- Map parts of code to required functionality
- Find out percentage of code covered by “instrumentation”
- Instrumented code provides fine control on execution and measuring it.
- Can help identify critical portions of code, executed often
- Coverage = (statements executed / Total statements) * 100

Types of Code Coverage

1. Statement coverage
2. Path coverage
3. Condition coverage
4. Function coverage

Programming Constructs

1. Sequential instructions
2. Two-way decision statements (if – then – else)
3. Multi-way decision statements (switch statements)
4. Loops, like while/do, for, repeat/until, etc

Testing Sequential Instructions

- Generate test data to make the program enter the sequential block, to make it go through the entire block
  - Asynchronous exceptions
  - Multiple entry points, in non-structured programming
- Statement coverage as a metric:
  - # of statements exercised / Total # of statements

Testing IF THEN ELSE

- Have data to test the THEN part
- Have data to test the ELSE part

- Similarly for multi-branch (switch) statements at least one test case for each case.

Coverage for Loop Constructs

- For, While, do-while, repeat..
- Typically cover boundary conditions w.r.t loop control expressions
  - 0 execution of loop
  - 1 execution
  - Max execution.., max-1

Limitation of Statement Coverage

```
Total = 0;                    /* set total to zero */
if (code == “M”) {
stmt1;
stmt2;
  stmt3;
  stmt4;
  stmt5; stmt6;stmt7
  }
  else
  percent = value/Total*100;  /* divide
by zero */
```

Code Coverage Tools

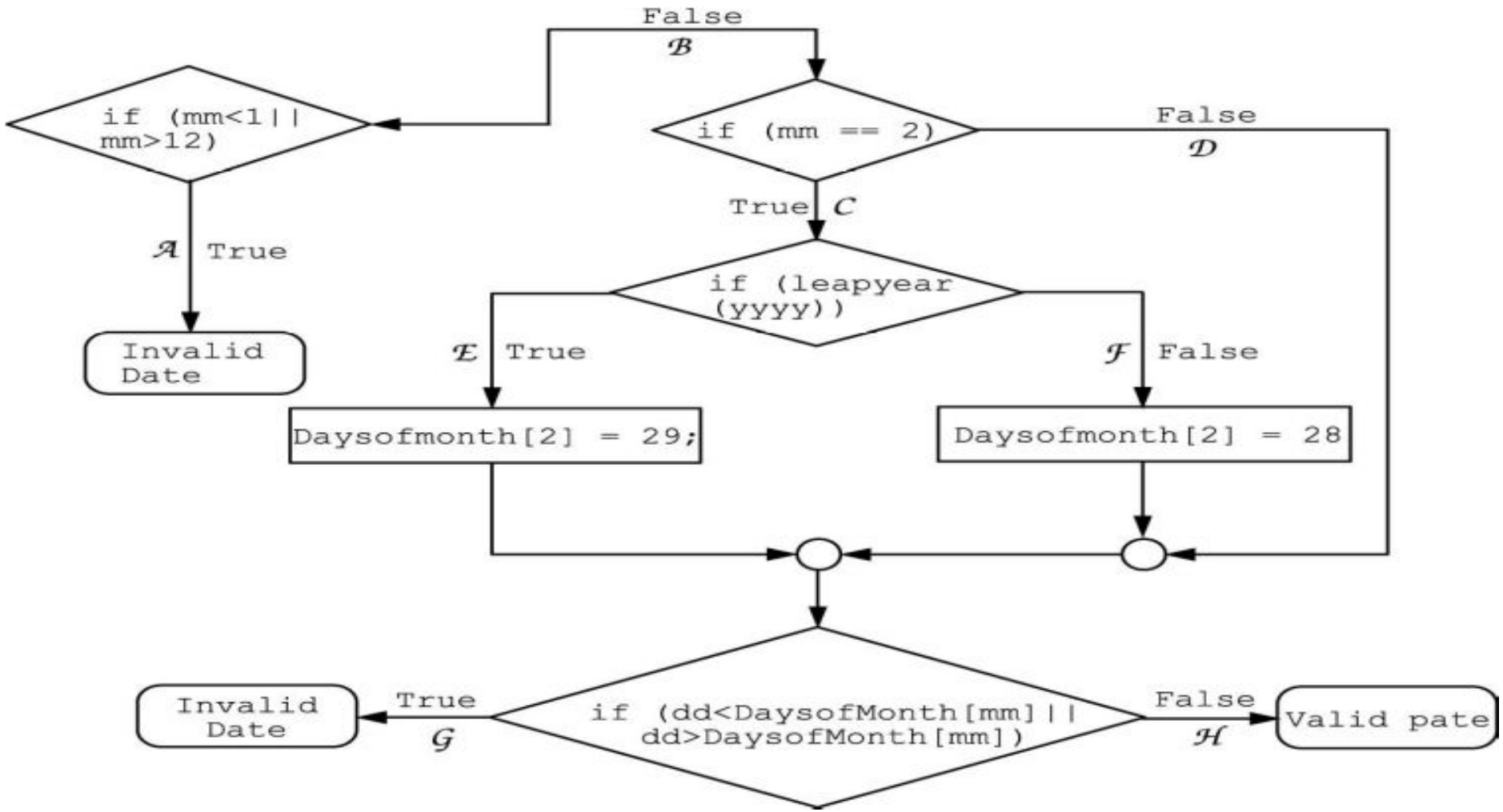
- Profilers – Jprofilers, Rational performance studio
- Assistance by compiler options
  - Symbol table preserving
  - Instrumentation
- Other tools
  - EMMA, Jcoverage, Clover

- Tools are strongly language dependant

Path Coverage

- Statement Coverage may not indicate "true coverage"
- Path Coverage provides better representation
  - Previous example: Having data go through the THEN part and ELSE part gives only 50% coverage, irrespective of number of statements in each "path"
- More detailed example of path coverage in next slide
- Path coverage = "# of paths exercised / total # of paths in the program"

Path Coverage.. Example
Date validation – mm dd yyyy



Condition Coverage

- Further refinement of path coverage
- Makes sure each constituent condition in a Boolean expression is covered
- Protects against compiler optimizations
- Condition coverage = # of conditions covered / number of conditions in the program

Function Coverage

- Finds how many functions are covered by test cases
- Higher level of abstraction; hence possible to achieve 100% coverage
- More “logical” than the other types of coverage
- Can be derived from “RTM”
- Easy to prioritize as they are based on requirements
- Leads more naturally to black box tests
- Can also be a natural predecessor to performance testing

Code Complexity Testing

- Which of the paths are independent? (If paths are not independent, we may be able to minimize the number of tests.)
- Is there an upper bound on the number of tests to be executed to ensure that all the statements have been executed at least once?
- Cyclomatic complexity provides answers to some of these questions.

Intuitive Meaning of Cyclomatic Complexity

- Provides an indication of how many “independent paths” are there in a program
- What gives birth to new paths?
- How can one count the new paths that are born?

Uses of Cyclomatic Complexity

- Helps developers and testers to determine independent path executions
- Developers can assure that all the paths have been tested at least once
- Helps us to focus more on the uncovered paths
- Improve code coverage in Software Engineering
- Evaluate the risk associated with the application or program
- Using these metrics early in the cycle reduces more risk to the program

https://www.guru99.com/cyclomatic-complexity.html

THANK YOU

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

Software Testing

Unit 2

Integration Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

List of Contents

Introduction

"We are lucky to find errors in our testing. Because, it is so much better than customer finding it"

What is Integration Testing?

- Integration: the act, or process or an instance of integrating
- Integrate: to form, coordinate or blend into a functioning or unified whole
- Typically a software system is made of several “blocks”. These blocks are “integrated” to form a whole system.
- Integration testing means testing a partially built system.
    - With the idea of ?

What is Integration Testing?

As Testing Type

- Based on Target - “ What”
- Testing of units / components/ modules after integrating them together – at any level
- Once integrated, appropriate technique (“How”) is used to test

As Testing Phase

- Since this activity is an important intermediate stage in STLC that needs special focus

Need for Integration Testing

- Integration testing uncovers inter-component interface and functional problems.
- Potential problems with external interfaces can be identified at earlier stage
- Since one deals with partially built system, some of the system-level problems could be discovered at earlier stage

Integration Testing - An Overview

- Concept is driven by
  - Software making process
  - Software complexity
  - Software/Technology solution Marketplace

Integration Testing - Errors

- Errors that are caught in Integration testing
  - Interface mismatch – Parameters / return value – order, type and other semantics
  - Missing interfaces
  - Service invocation protocol mismatch
  - Error handling across the interfaces
  - Errors due to deployed environment
  - Initializations, closures
  - Configuration errors
    - Errors related to “emergent properties” are not caught this way.

Integration Testing - Errors

1. Error manifestations
    - Direct system messages
        - “… Not found”, “… wrong parameter…”
        - “..Contact System administrator”
    - Functional misbehavior
2. Run-time errors

1. May not manifest / detect also

Integration Testing - Testing of Interfaces

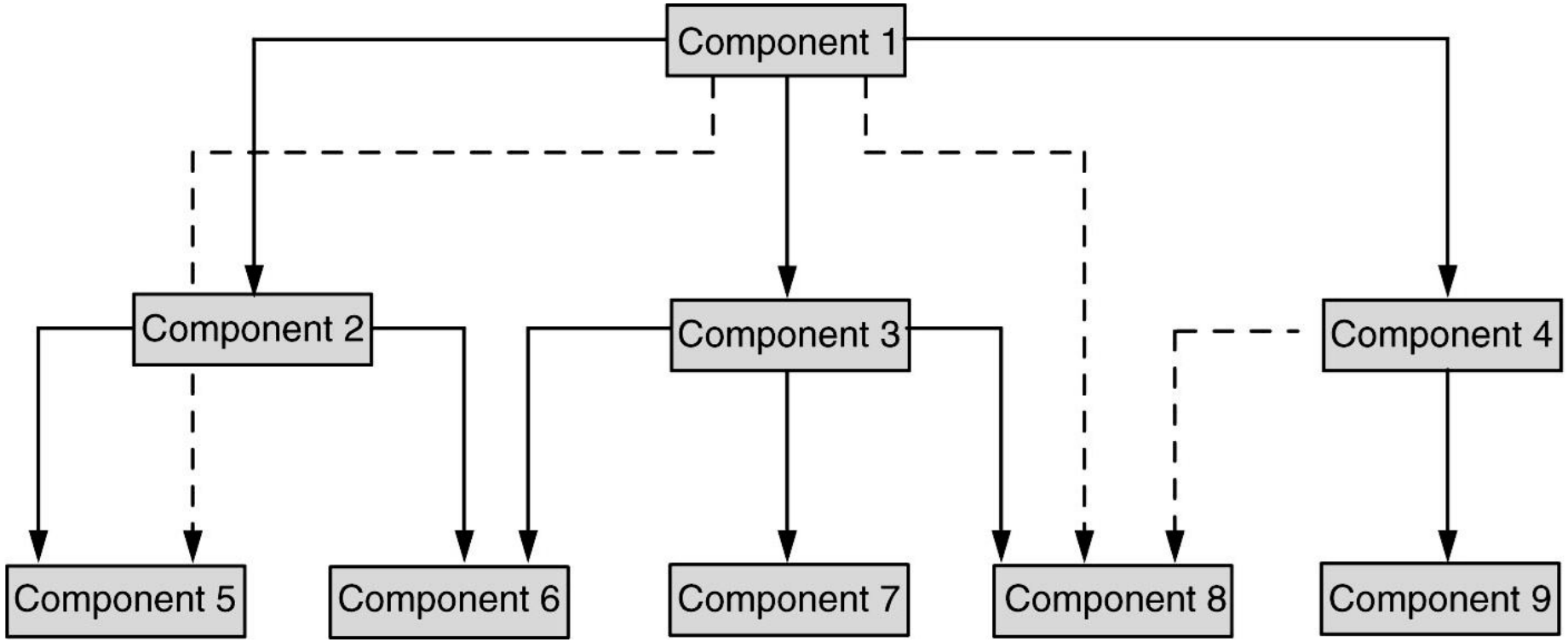
Internal interfaces

- The interfaces that are provided for use by other devs / components
- Provides communication across two or more modules
- Not exposed to the customers
- Needs complete understanding of design & architecture

External interfaces

- The interfaces provided for use by other system/products
- Used by third-party developers, solution providers
- You need to understand why they are provided and how they can be used
- APIs / SDKs / scripts / queries / dynamically created scripts / Web services

Integration Testing - Testing of Interfaces



Integration Testing - Factors

Integration sequence depends on a number of factors:

- Test strategy
- Project plan compulsions
- Dev methodology – agile / spiral…
- Actual project progress
- External / customer dependency
- Testing Resource availability
- Testing method?

Integration Testing - Methods

A systematic approach

1. **Top down**
2. **Bottom up**
3. **Bi-Directional**
4. **System Integration**

All are incremental – except System Integration

Stubs & Drivers

Top Down Integration Testing – Stubs:

Stubs are program units that are stand-instruction for the other(more complex) program units that are directly referenced by the unit being tested

- It provides an **Interface that is identical** to the interface that will be provided by the actual program unit, and the minimum acceptable behavior expected of the actual program unit (a return statement in a function)

- Very much needed to test the dependent program units till this unit is not ready

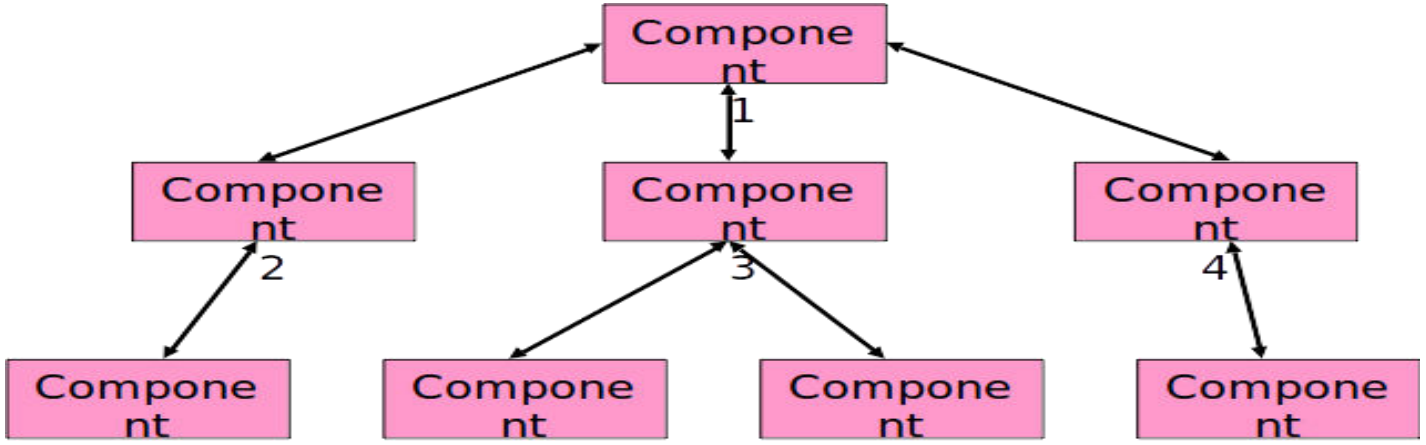
Bottom Up Integration Testing – Drivers:

Drivers are programs or tools that allow a tester to exercise/examine in a controlling manner the unit of software being tested.

- It provides a means of defining, declaring or otherwise creating, any variables, constants or other items needed in the testing of the unit and a means of monitoring the states of these items, any input and output mechanisms needed in the testing of unit.

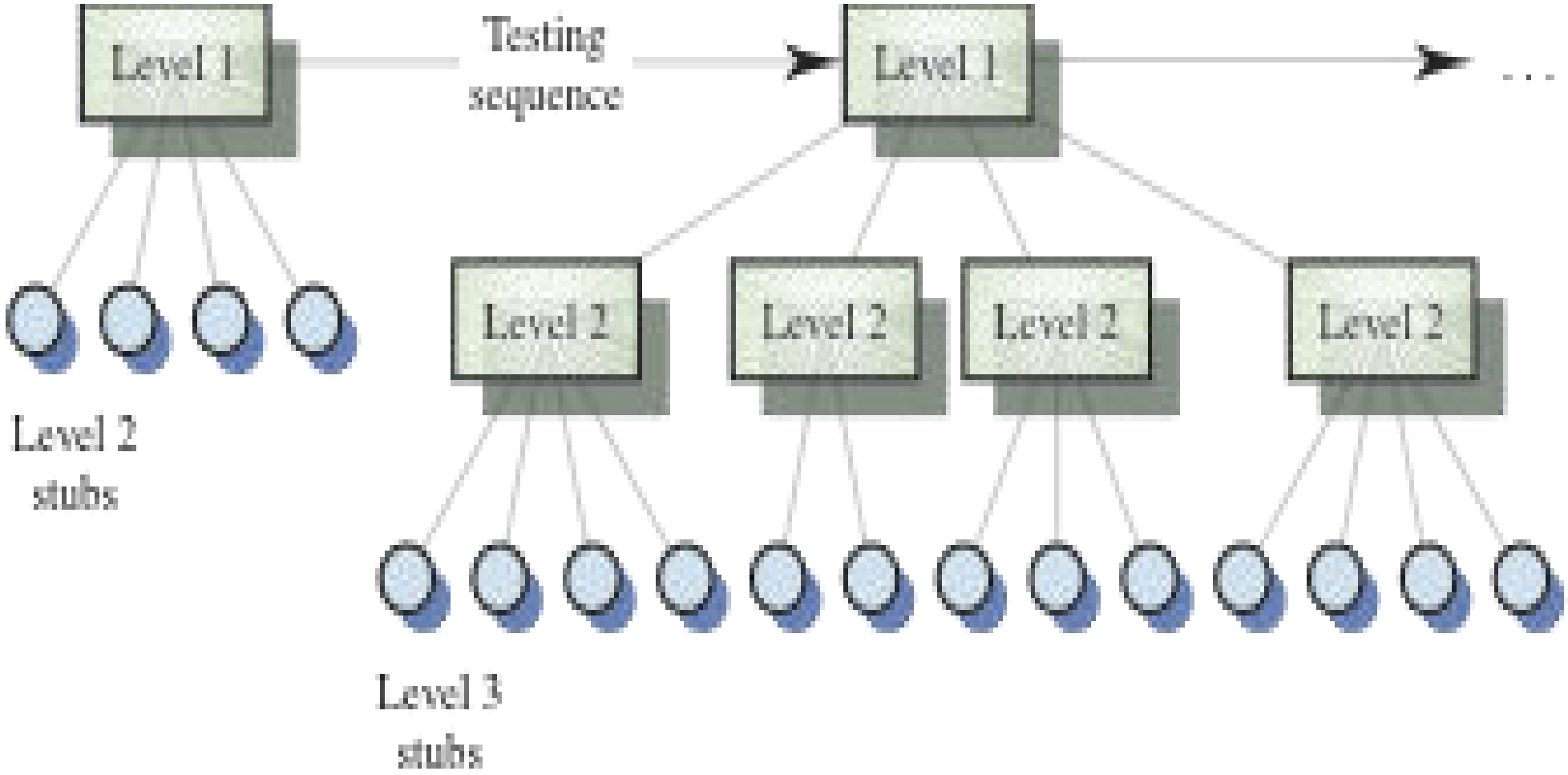
- “Test Harnesses”

Top Down Integration



Step	Interfaces Tested
1	1-2
2	1-3
3	1-4
4	1-2-5
5	1-3-6
6	1-3-6-(3-7)
7	(1-2-5)-(1-3-6-(3-7))
8	1-4-8
9	(1-2-5)-(1-3-6-(3-7))-(1-4-8)

Highest Levels are Integrated First



Issues in Top Down Integration

- If some unit/component not available, “stub” required
- Some parts may be treated as sub-system
- Breadth first – depth first

Top Down Integration

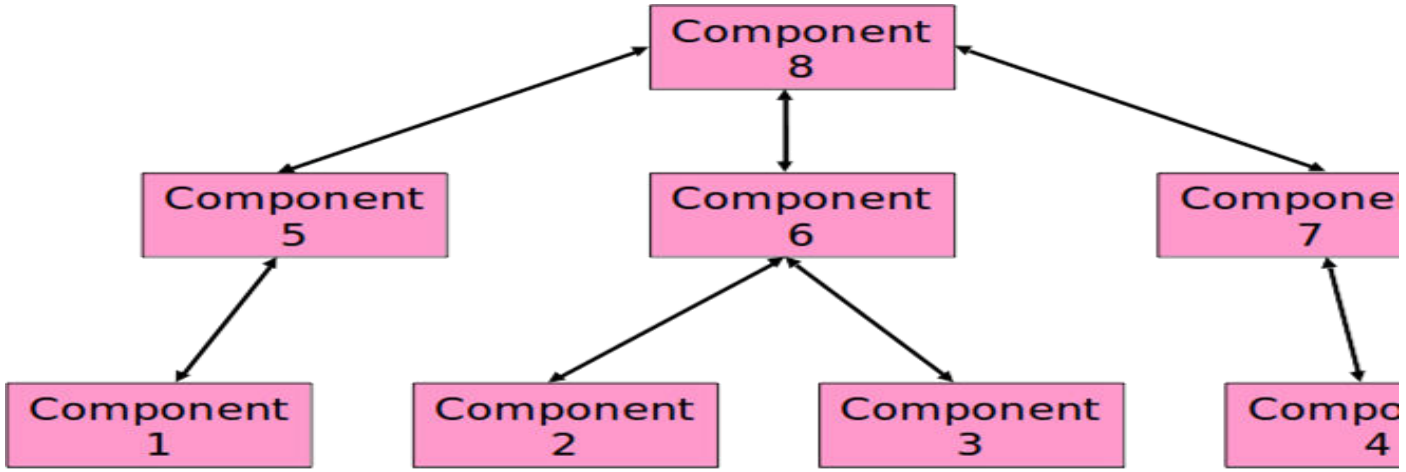
Advantages:

- High-level logic and data flow is tested early in the process.
- It tends to minimize the need for ***drivers***.
- Early skeletal program allows demonstration of click-thru prototype and boosts morale of developers and end users.
- The coverage of test cases improve without any changes as new modules are added at the bottom.

Disadvantages:

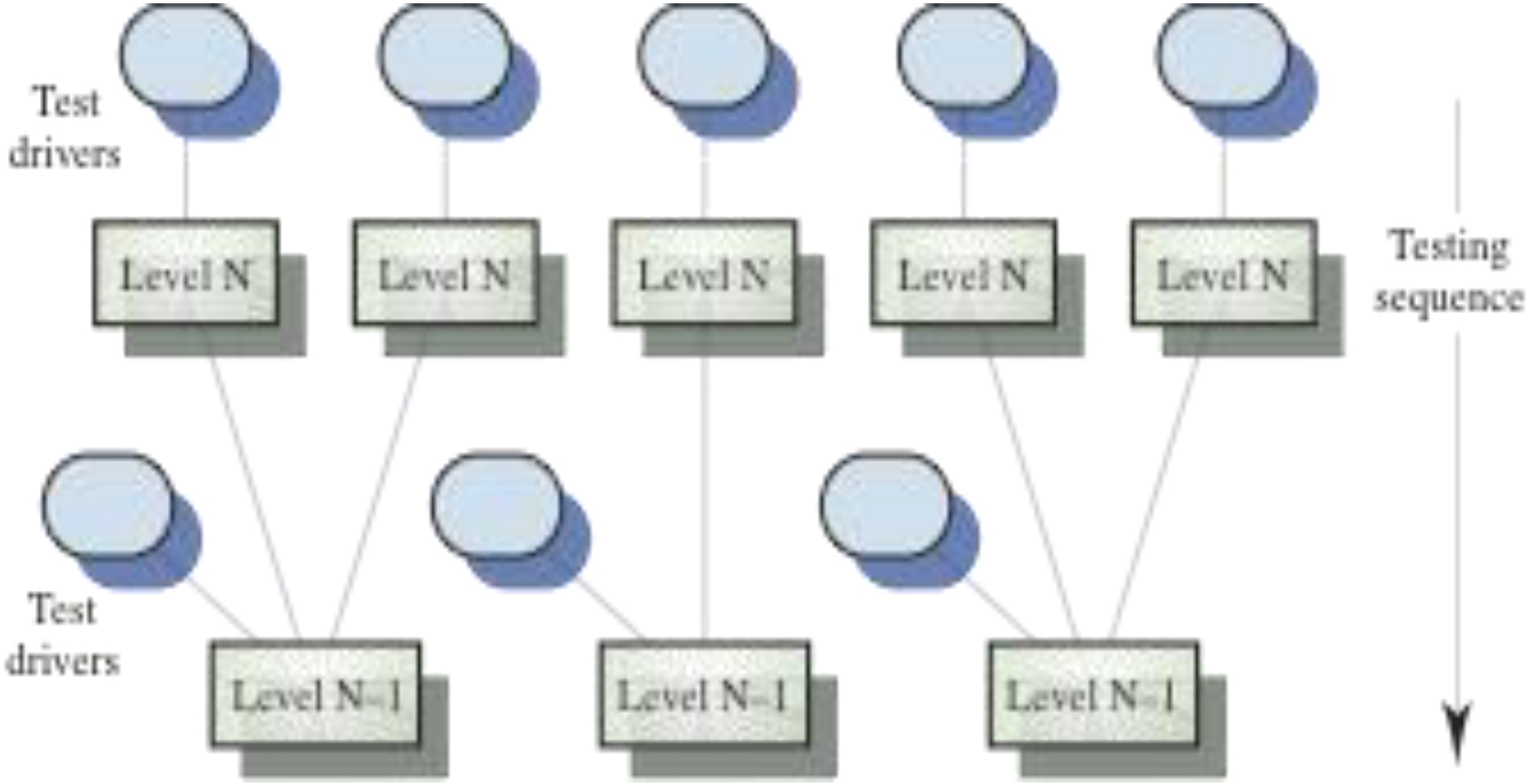
- Need for ***stubs*** complicates testing effort.
- Difficult to observe the flow of data.
- Low-level utilities are tested relatively late in the development cycle.
- Since low level functions are not needed upfront, they may be designed at later stage which is not a good practice.
- Poor support for early release of limited functionality.

Bottom Up Integration



Step	Interfaces Tested
1	1-5
2	2-6, 3-6
3	2-6-(3-6)
4	4-7
5	1-5-8
6	2-6-(3-6)-8
7	4-7-8

Bottom Up Integration



Issues in Bottom Up Integration

- Quite suited for Agile kind of methodologies
- If some unit/component not available, “stub” required
- From quality perspective, better approach as the ‘leaf nodes’ get tested many times.

Bottom Up Integration

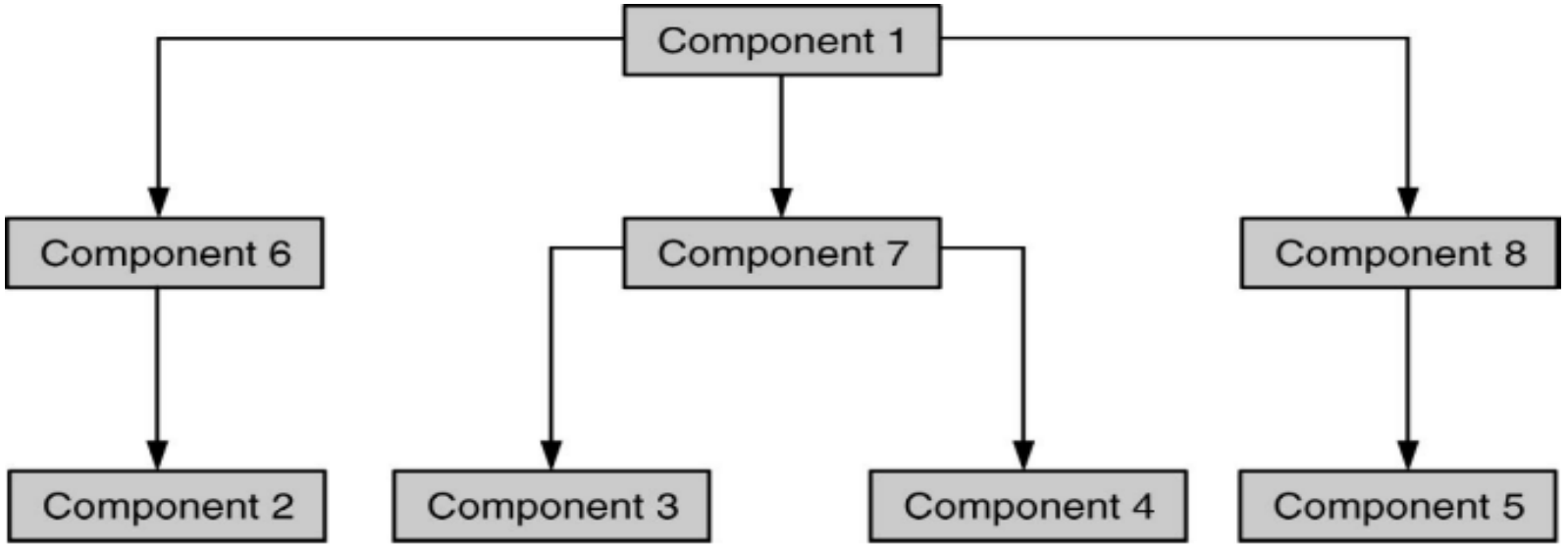
Advantages:

- Utility modules are tested early in the development process
- Need for **stubs** is minimized
- It is easier to observe the flow of data

Disadvantages:

- Need for **drivers** complicates testing efforts
- High-level logic and data flow are tested late
- A demonstrable program can be created only after last (top) module is added
- Many loose integration segments to be managed
- Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality

Bidirectional Integration



Step	Interfaces Tested
1	6-2
2	7-3-4
3	8-5
4	(1-6-2)-(1-7-3-4)-(1-8-5)

Bidirectional Integration

- Combination of Top-down & Bottom up.
- Individual components are separately tested.
- Bi-Direction integration is done with use of Drivers & Stubs.
- Drivers - Upstream connectivity.
- Stubs - Downstream connectivity.
- Drivers & stubs are discarded after integration testing.
- Sandwich Integration - is the combination of bottom-up approach and top-down approach, so it uses the advantage of both bottom up approach and top down approach. Initially it uses the stubs and drivers where stubs simulate the behaviour of missing component.

Incremental Integration

Advantages:

- Programming errors related to mismatching interfaces or incorrect assumptions among modules will be detected earlier.
- Debugging is easier.
- It offers more flexibility.
- It enables more efficient use of resources.
  - Both technically and managerially this is preferred to “big-bang” /System level

System or “Big-Bang” Integration

System integration means that all the components of the system are integrated together and tested as an entire unit.

Disadvantages:

- When a failure or defect is encountered during system integration test, it is very difficult to locate the problem.
- The ownership for correcting the root cause of the defect may be an issue difficult to pinpoint.
- When integration testing happens in the end, the pressure for approaching release date is very high and may affect quality.
- A certain component may take excessive amount of time to get ready. This precludes testing other interfaces and integration testing waits till the end.

THANK YOU

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

27

Software Testing

Unit 2

Scenario Testing & System Integration

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

List of Contents

Scenario Testing

- Software Testing Technique that uses scenarios i.e. speculative stories to help the tester work through a complicated problem or test system.
- The ideal scenario test is a reliable, complicated, convincing or motivating story the outcome of which is easy to assess.
- Usually these tests are different from test cases as the test cases are single steps whereas scenarios cover a number of steps.
- Scenario testing is carried out by creating test scenarios which copy the end users usage.
- A test scenario is a story which describes the usage of the software by an end user.

- Example: Hospital Management System
https://softwaretestingmentor.com/what-is-scenario-testing/
- Ecommerce application
- Test scenarios for a Banking Site
https://www.guru99.com/test-scenario.html

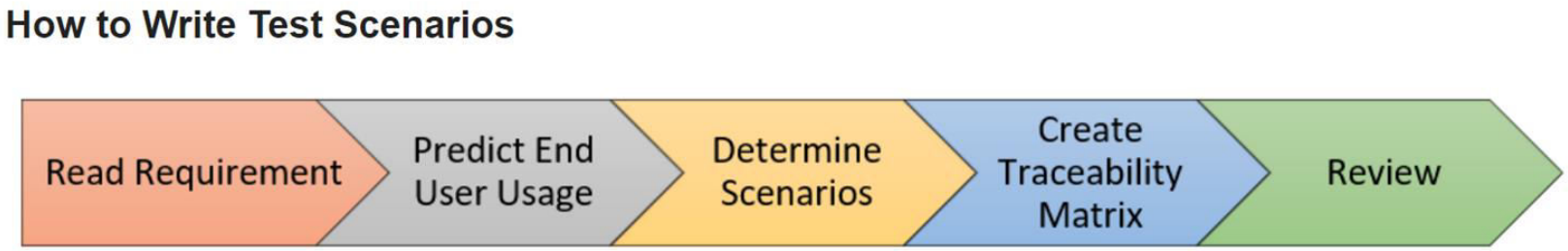
Scenario Testing

❖A scenario test has five key characteristics:
✔Story
✔Motivating
✔Credible
✔Complex
✔Easy to evaluate

Risks of Scenario Testing:

- Scenario testing is complex involving many features.
- Scenario testing is not designed for coverage of the program or for test coverage.
- Scenario testing is often heavily documented and used time and again.
- When the product is unstable, scenario testing becomes complicated.

Scenario Testing



Step 1: Read the Requirement Documents like BRS, SRS, FRS, of the System Under Test (SUT).
Step 2: For each requirement, figure out possible users actions and objectives. Determine the technical aspects of the requirement.
Step 3: After reading the Requirements Document and doing your due Analysis, list out different test scenarios that verify each feature of the software.
Step 4: Once you have listed all possible Test Scenarios, a Traceability Matrix is created to verify that each & every requirement has a corresponding Test Scenario
Step 5: The scenarios created are reviewed by your supervisor. Later, they are also reviewed by other Stakeholders in the project.

Test Scenarios for a Banking Site (Example)

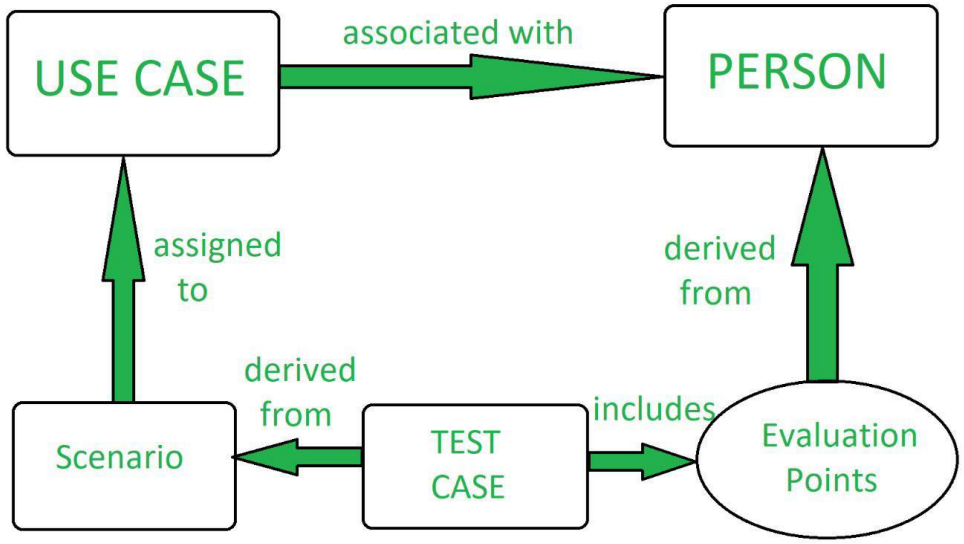
Test Scenario 1: Check the Login and Authentication Functionality
Test Scenario 2: Check Money Transfer can be done
Test Scenario 3: Check Account Statement can be viewed
Test Scenario 4: Check Fixed Deposit/Recurring Deposit can be created

Methods in Scenario Testing

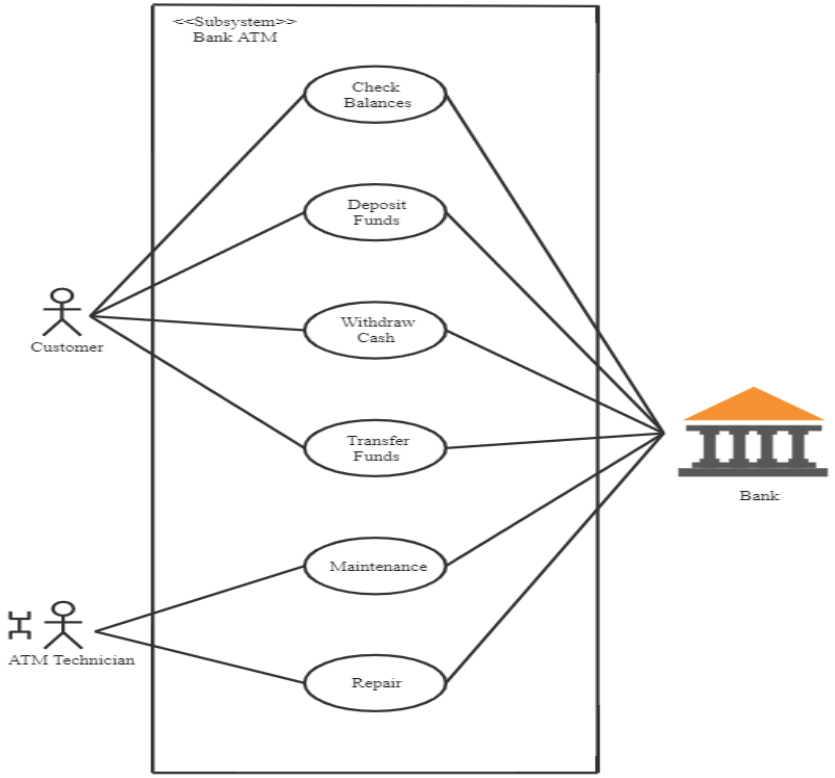
SYSTEM SCENARIOS : Scenario tests used in this method are only those sets of realistic, user activities that cover various components in the system. Story Line, Life Cycle/state transitions, Deployment/Implementation stories from customer, Business verticals, Battle ground

USE CASE SCENARIOS: In use-case and role-based scenario method the focus is specifically on how the system is used by a user with different roles and environment.

Methods in Scenario Testing Example



(I) SYSTEM SCENARIO



(II) USE CASE SCENARIO

Use Case Scenario (Integration Testing)

Actor	System Response
User would like to withdraw cash and inserts the card in ATM machine	Request for Password or PIN (Personal identification number)
User Fill-in the Password or PIN	
	Validate the password or PIN
	Give a list containing types of accounts
User selects an account type	
	Ask the user for amount
User Fill-in the amount of cash required	
	Check availability of funds Update account balance Prepare receipt Dispense cash
Retrieve cash from ATM	
	Print receipt

Strategies to Create Good Scenarios

- Enumerate possible users their actions and objectives
- Evaluate users with hacker's mindset and list possible scenarios of system abuse.
- List the system events and how does the system handle such requests.
- List benefits and create end-to-end tasks to check them.
- Read about similar systems and their behaviour.
- Studying complaints about competitor's products and their predecessor.
- For each requirement, figure out possible users actions and objectives.
- Determine the technical aspects of the requirement.
- Ascertain possible scenarios of system abuse and evaluate users with hacker’s mindset.

Scenario Testing (End to End Testing)

- Testing “full aspect of main functionality” – A business process for example.
- Ex. Order processing:
- Create customer, create product, collect customer order, prepare making/packing, dispatch, collect money
- Focus is on ensuring the full function works and the associated building blocks are integrated well.
- This the first step to do full functional test with detailed sub-steps.

System Integration

System integration means that all the components of the system are integrated together and tested as an entire unit.

Advantages:

- Saves time & effort.

Disadvantages:

- When a failure or defect is encountered during system integration test , it is very difficult to locate the problem.
- The ownership for correcting the root cause of the defect may be an issue difficult to pinpoint.
- When integration testing happens in the end, the pressure for approaching release date is very high and may affect quality.
- A certain component may take excessive amount of time to get ready. This precludes testing other interfaces and integration testing waits till the end.

System Integration

- System integration means that all the components of the system are integrated together and tested as an entire unit.

- 2 types :
  - Sub System Integration
  - Final/ system integration

Big Bang Integration – approach in which all software components (modules) are combined at once and make a complicated system. This unity of different modules is then tested as an entity.
Ideal for a product where the interfaces are stable and less number of defects.

Selecting Integration Method

S.No	Factors	Suggested Integration method
1	Clear requirements and design	Top down
2	Dynamically changing requirements, design, architecture	Bottom up
3	Changing architecture, stable design	Bi-directional
4	Limited changes to existing architecture with less impact	System
5	Combination of above	Select one of the above after careful analysis.

Steps in Integration Testing

Step 1: Create Test Cases and Test Data
Step 2: Create a Test Plan
Step 3: Once the planned components have been integrated, setup along with environment
Step 4: Execute the test cases
Step 5: Report results
Step 6: Repeat steps 4 & 5 when required – if defects
Step 7: Repeat the above test until all the components have been successfully integrated
Other: Use automation – tools/scripts – where appropriate

THANK YOU

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

Software Testing

Unit 2

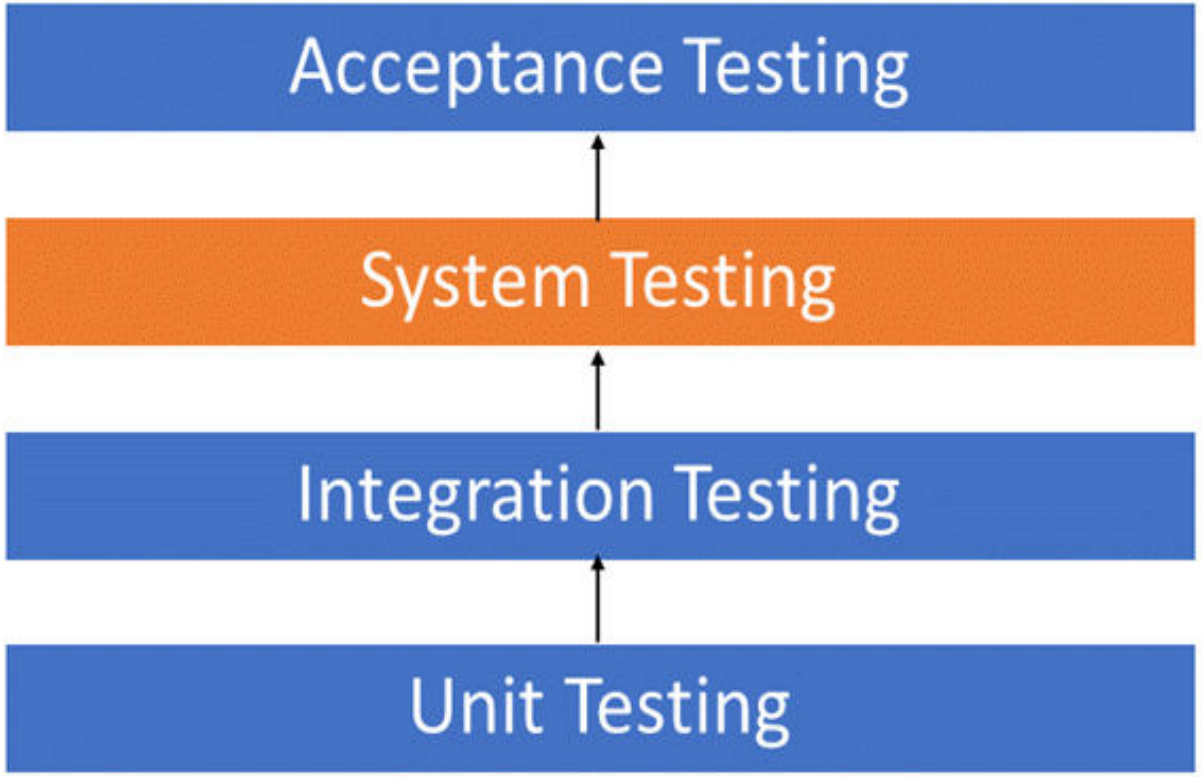
System Testing

Prof Raghu B. A. Rao

Department of Computer Science and Engineering

List of Contents

System Testing



System Testing

- System testing tests a completely integrated system to verify that its compliant with its specified requirements.
- It seeks to detect defects both within the "inter-assemblages" and also within the system as a whole
- Testing in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS).
- Focus is functionality and performance and not negative / exception cases (which are already done by now)

“System testing is the only phase of testing which tests both functional and non-functional aspect of the product”

Functional : real time usage if product

Non Functional : System brings different type of testing

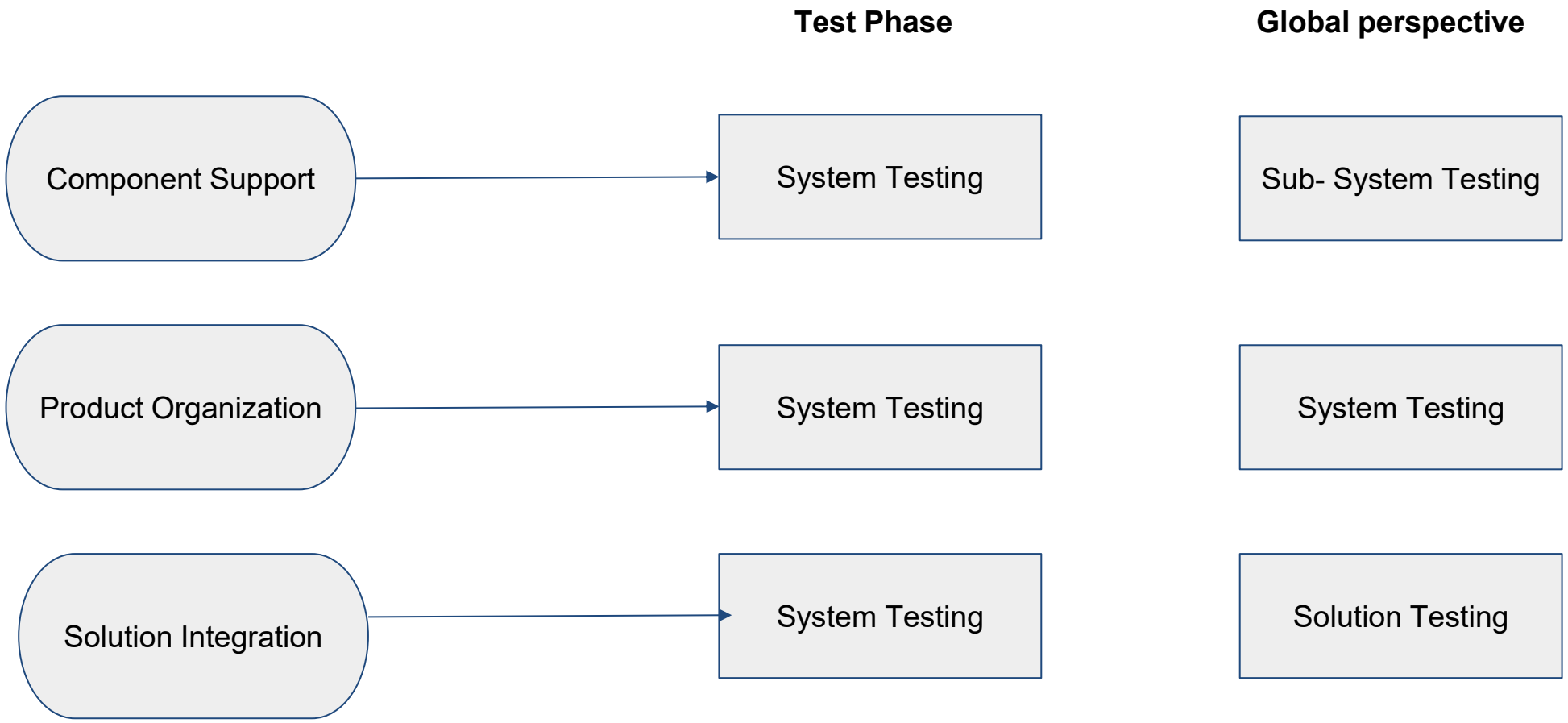
System Testing (End-to-End Testing)

- Testing “full aspect of main functionality” – A business process for example.
- Create scenarios
- **Story-line** : Develop a story-line that combines various activities of the product
- **Life-cycle / state transitions**: Consider an object, derive the different transitions / modification that happen to the object and derive scenarios to cover them
- **Deployment / implementation details from customer**
- **Business verticals**: Visualizing how a product / software will be applied to different business verticals and create a set of activities as scenarios (e.g., insurance, life sciences)
- **Battle-ground scenarios**: Create some scenarios to justify “the product works” and some scenarios to “try and break the system” to justify “the product doesn’t work.”

Why is System Testing Done?

1. Provide independent perspective in testing.
2. Bring in customer perspective in testing.
3. Provide a fresh pair of eyes to discover defects not found earlier by testing.
4. Test product behaviour in a holistic, complete and realistic environment.
5. Test both functional and non-functional aspects of the product.
6. Build confidence in the product.
7. Analyze and reduce the risk of releasing the product
8. Ensure all requirements are met and ready the product for acceptance testing.

Different Perspectives of System Testing



Functional Testing

- Type of software testing that validates the software system against the functional requirements/specifications.
- The purpose of Functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the Functional requirements.
- Functional testing mainly involves black box testing and it is not concerned about the source code of the application.
- This testing checks User Interface, APIs, Database, Security, Client/Server communication and other functionality of the Application Under Test.
- The testing can be done either manually or using automation.

Non Functional Testing

- Type of Software testing to check non-functional aspects (performance, usability, reliability, etc) of a software application.
- It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.
- Example - check how many people can simultaneously login into a software.
- Non-functional testing is equally important as functional testing and affects client satisfaction.

What is the functional requirement of this milk cartoon?

What is the non functional requirement of this hard helmet

Testing Aspects	Functional Testing	Non Functional Testing
Involves	Product feature and functionality	Quality factors
Tests	product Behavior	Behavior and experience
Result Conclusion	Simple steps written to check expected results	Huge data collected and analysed
Results varies due to	Product Implementation	Product Implementation, resources and configuration
Testing Focus	Defect Detection	Qualification of Product
Knowledge Required	Product and domain	Product, domain, design,architect, statistical skill.
Failure is normally due to	code	architecture, design and code
Testing phase	Unit, component, integration system	System
Test case repeatability	Repeated many times	Repeated only in case of failures and different configurations
Configuration	One time set up for a set of test cases	Configuration changes for each test case.

Types of Non-Functional Testing

1. **Performance & Load Testing** - To evaluate time taken of system to perform its required functions in comparison with different versions of same product or different competitive product is called performance testing.

1. **Scalability Testing** - A testing that requires enormous amount of resource to find out the maximum capability is the system parameters.

1. **Reliability Testing** - Checks whether the software can perform a failure-free operation for a specified time period in a particular environment.

1. **Stress Testing** - Evaluating the system beyond the limits or specified requirements to ensure the system does not break down unexpectedly.

1. **Interoperability Testing** - Testing is done to ensure that two or more products can exchange information.

1. **Localization Testing** - Testing conducted to verify that the localized product works in different languages.

Design/Architecture Verification

- Test cases are developed and checked against the designs and architecture to see whether they are actual product level test cases.

- Functional test cases focuses on the behaviour of the entire product. This techniques helps to validate the product features that are written based on the customer scenarios.

Business Vertical Testing

Business vertical testing is a process in which a product is being used and tested for different business verticals such as banking, insurance, asset management, etc., and also verifying the business operations and its usage. The two types are:

- **Simulation** – In the simulation of a business vertical test, the customer or the tester assumes requirements, and the business flow is tested.

- **Replication** – In the replication process, the client data and process are acquired and the product is fully customized, and tested, and the customized product as it was tested is released to the client.

Beta Testing

Beta testing is a type of **User Acceptance Testing**.

Testing of the product performs by the **real users of the software** application in the real environment. Beta version of the software is released to a **restricted number of end-users** to obtain the feedback of the product quality.

Beta testing reduces the risk of failure and provides the quality of the product through customer validation.

THANK YOU

Prof Raghu B. A. Rao

Department of Computer Science and Engineering