LECTURE 7

Software requirements Engineering

Some basics about requirement and testing

Instructor:
Dr. Natalia Chaudhry,
Assistant Professor, PUCIT, University of the Punjab, Lahore.

Outline

- Testing?
- Defects?
- Baselines
- Test cases
- Test suites
- Test coverage and requirement coverage
- Test driven development

- Software testing is undoubtedly the largest consumer of software quality assurance resources
- “Testing is the process of executing a program with intention of finding errors.”

Defects refer to any flaws or issues identified in the software that prevent it from functioning correctly or meeting its requirements. These defects are also commonly referred to as bugs or issues.

Testing activities help in discovering and resolving defects before software deployment.

Baselines

- In software testing, a baseline refers to a set of documentation, requirements, or artifacts that serves as a reference point for future comparison.
- It represents a stable and approved version of a document, system, or component against which changes or deviations can be measured.
- Baselines are commonly used to establish a starting point for testing activities and to track changes throughout the software development lifecycle.

Types of Baselines:

- **Document Baseline**: This includes documents such as requirements specifications, design documents, test plans, and other project documentation.
- **Configuration Baseline**: This includes the configuration items (e.g., code, databases, libraries) that make up the software system.
- **Product Baseline**: This represents a stable version of the software product itself, usually after completion of a development phase or milestone.

- Baselines provide a <u>**reference point**</u> for comparison to track changes and ensure consistency throughout the development and testing process.
- Baselines help in <u>**managing changes**</u> by providing a standard against which proposed changes can be evaluated and approved.

Scenario: A software development team is tasked with developing a new mobile application for a ride-sharing service. The application will allow users to request rides, track their drivers in real-time, make payments, and provide feedback on their experience. The project is divided into several phases, including requirements gathering, design, development, testing, and deployment.

Requirements Baseline:

The development team collaborates with stakeholders to gather requirements for the ride-sharing app, including features, user stories, and business rules.

Baseline Usage: The collected requirements are documented and approved by stakeholders, establishing the requirements baseline for the project.
Example: The requirements baseline includes features such as user registration, ride booking, real-time tracking, payment integration, and driver ratings.

Design Baseline:

Based on the approved requirements, the design team creates wireframes and designs the user interface (UI) and user experience (UX) for the ride-sharing app.

Baseline Usage: The UI/UX designs are reviewed and finalized, serving as the design baseline for the development phase.
Example: The design baseline includes mockups of the app's home screen, ride request interface, map view for tracking drivers, payment flow, and feedback submission form.

Code Baseline:

Developers begin coding the ride-sharing app based on the finalized designs and requirements.

Baseline Usage: As code is written and integrated into the version control system (e.g., Git), each stable release represents a code baseline.
Example: The code baseline includes implementation of features such as user authentication, location tracking, ride request handling, payment processing, and integration with third-party services.

Test Baseline:

QA engineers develop test cases and test plans based on the requirements and design baselines to ensure the functionality and quality of the ride-sharing app.

Baseline Usage: The initial version of the app is tested against the test baseline to identify and fix defects.

Example: The test baseline includes test cases for functionality like user registration, ride booking, driver availability, payment processing, and feedback submission.

Product Baseline:

After successful testing and bug fixing, the ride-sharing app is ready for deployment to users.

Baseline Usage: The deployed version of the app represents the product baseline, meeting the approved requirements and quality standards.
Example: The product baseline includes the live version of the ride-sharing app accessible to users, with all agreed-upon features and functionalities.

Test cases

- A test case is a specific set of conditions or variables under which a tester will determine whether a system, application, or feature is working correctly or not.
- It consists of inputs, execution conditions, and expected results.
- Test cases are created based on requirements, specifications, or user stories to verify that the software behaves as intended.
- They are an essential part of the software testing process and help ensure the quality and reliability of the software being developed.

Components:

- **Test Case ID**: A unique identifier for each test case.
- **Test Case Description**: A brief description explaining what the test case is intended to test.
- **Test Steps**: Detailed instructions outlining the sequence of actions to be performed during the test.
- **Test Data**: The input values or conditions necessary to execute the test case.
- **Expected Results**: The anticipated outcome or behavior of the software when the test case is executed successfully.
- **Actual Results**: The observed outcome or behavior of the software during test execution.
- **Pass/Fail Status**: Indicates whether the test case passed or failed based on a comparison between the actual and expected results.

# EXAMPLE Verify addition functionality of the calculator.

```
def test_addition():
    # Test Data
    num1 = 5
    num2 = 7

    # Expected Result
    expected_result = 12

    # Actual Result
    actual_result = add(num1, num2)

    # Assertion

# Function to perform addition
def add(a, b):
    return a + b

# Test execution
test_addition()
```

We use an assertion to compare the actual_result with the expected_result. If they are not equal, the test case will fail, and an error message will be displayed.

This is handled by a tool/library. For now, ignore it!

some example test cases for the ride-sharing app

User Registration:
Test Case 1: Verify that the user can register with valid credentials (email, password).
Test Case 2: Verify that the user cannot register with invalid or duplicate email addresses.
Test Case 3: Verify that the password meets the specified criteria (e.g., minimum length, special characters).

Ride Booking:
Test Case 4: Verify that the user can search for available rides by entering the pickup and drop-off locations.
Test Case 5: Verify that the user can select a ride option (e.g., car type, ride-sharing vs. solo ride).
Test Case 6: Verify that the user receives a confirmation notification after booking a ride successfully.

Driver Availability:
Test Case 7: Verify that drivers are displayed on the map based on their availability and proximity to the user's location.
Test Case 8: Verify that the user can see the estimated time of arrival (ETA) for each available driver.
Test Case 9: Verify that the user can choose a preferred driver from the available options (if applicable).

Payment Processing:
Test Case 10: Verify that the user can add and save payment methods (credit/debit cards, PayPal, etc.).
Test Case 11: Verify that the user can select a payment method and complete the payment process for the booked ride.
Test Case 12: Verify that the user receives a payment confirmation and receipt after successful payment.

Ride Tracking:
Test Case 13: Verify that the user can track the location of the assigned driver in real-time on the map.
Test Case 14: Verify that the user receives updates on the driver's ETA and location throughout the ride.
Test Case 15: Verify that the user can contact the driver (e.g., call or message) if needed during the ride.

Feedback Submission:
Test Case 16: Verify that the user can provide feedback and ratings for the completed ride.
Test Case 17: Verify that the user can submit feedback for both the driver and the overall ride experience.
Test Case 18: Verify that the feedback is successfully recorded and reflected in the driver's profile and ratings.

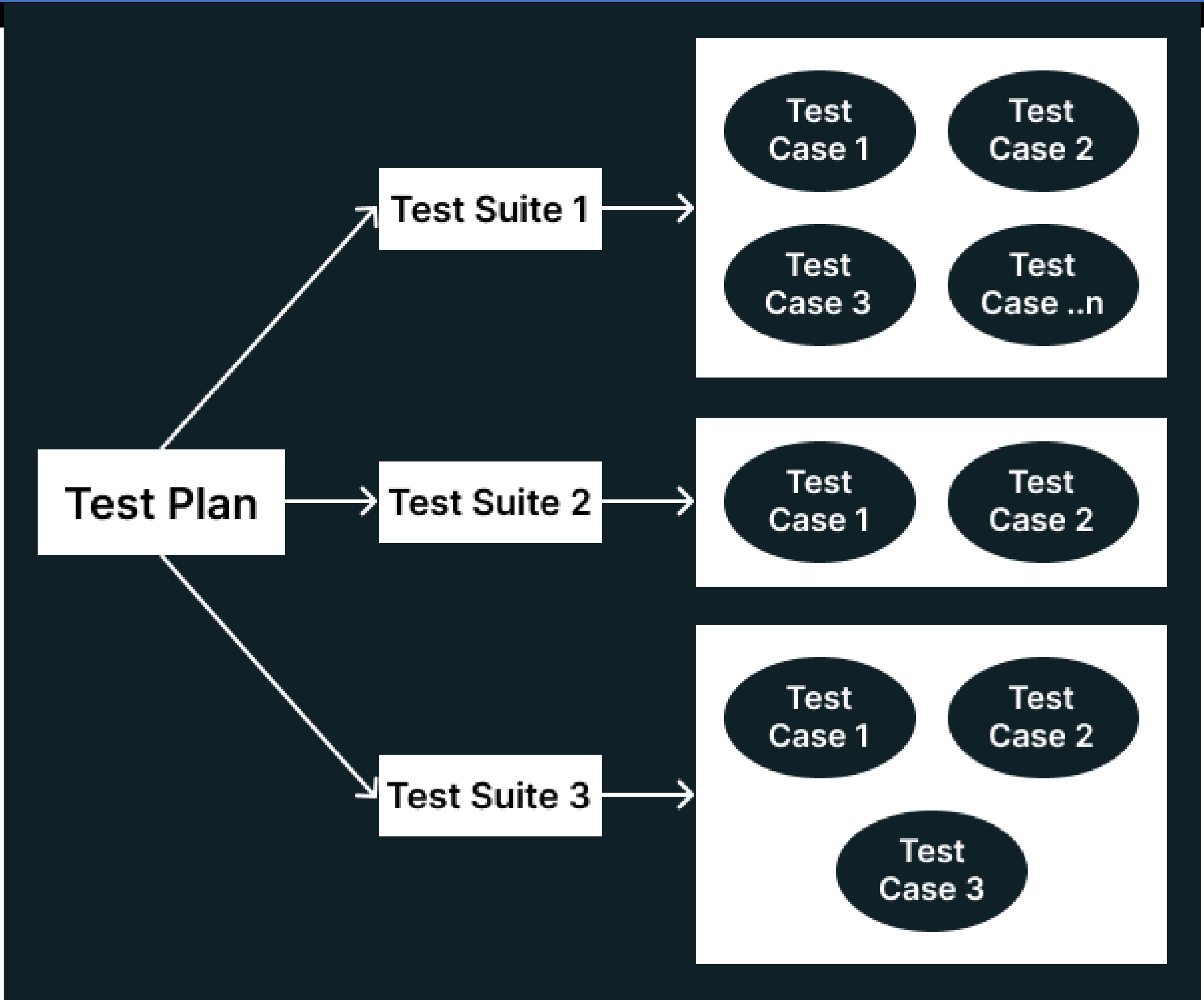
Test suites

Test suites are the logical grouping or collection of test cases to run a single job with different test scenarios

For instance, a test suite for product purchase has multiple test cases, like:

- Test Case 1: Login
- Test Case 2: Adding Products
- Test Case 3: Checkout
- Test Case 4: Logout

A test suite also acts as a container for test cases.


Test Plan
Test Suite 1
Test Suite 2
Test Suite 3
Test Case 1
Test Case 2
Test Case 3
Test Case ..n
Test Case 1
Test Case 2
Test Case 1
Test Case 2
Test Case 3

Test and requirement coverage

Test coverage defines what percentage of application code is tested and whether the test cases cover all the code. It is calculated as

- $test\ coverage = \frac{lines\ of\ code\ covered\ by\ tests}{total\ lines\ of\ code} * 100$

- In the Requirements module, you create test coverage by linking tests to a requirement.
- Creating test plans based on requirements not only helps you in designing test cases but also ensures that all requirements are covered at the stage for formulating.
- The goal is to ensure that each requirement is associated with at least one test case, and ideally, that all requirements are thoroughly tested to ensure comprehensive coverage.

- Verifying the coverage of each requirement can be a time-consuming task especially when you’re working with a product that contains hundreds or thousands of requirements, tasks and sub-tasks.
- With the help of the **requirement traceability matrix**, you can quickly view the particular requirement and their linked requirements, as well as access information related to the status of a given requirement and test case.

Requirement: The system shall allow users to search for books by title, author, or genre.

Test Cases:

1. Enter a book title and verify that relevant search results are displayed.
2. Enter an author's name and verify that books by that author are displayed.
3. Select a genre from the dropdown menu and verify that books belonging to that genre are displayed.
4. Enter a keyword that does not match any book and verify that no results are displayed.

Requirement Traceability Matrix (RTM):

Requirement ID	**Requirement Description**	**Test Case ID(s)**
REQ-001	Allow users to search by title	TC-001
REQ-002	Allow users to search by author	TC-002
REQ-003	Allow users to search by genre	TC-003
REQ-004	Display relevant search results	TC-001, TC-002, TC-003, TC-004

ToDo

User Profile Management:

REQ-001: Users must be able to update their profile information.

REQ-002: Users must be able to upload a profile picture.

Test Cases:

User Profile Management:

TC-001: User updates their name.

TC-002: User updates their email address.

TC-003: User updates their phone number.

TC-004: User uploads a profile picture.

Requirement ID	Requirement Description	Test Case ID(s)
REQ-001	Users must be able to update their profile information	TC-001, TC-002, TC-003
REQ-002	Users must be able to upload a profile picture	TC-004

Test-driven development (TDD)

- Test-driven development (TDD) is a software development methodology where the development process is driven by writing tests for the software before writing the code itself.
- The key idea behind TDD is that by writing tests first, developers are forced to think about the requirements and design of the software before writing any code.
- This helps in producing more reliable and maintainable code
- Additionally, having a comprehensive suite of tests helps in identifying and fixing bugs early in the development process.

The cycle typically follows these steps:

Write a Test: First, the developer writes a test that defines a small part of the desired functionality of the software. This test will initially fail because the corresponding code has not yet been written.

Write the Code: Next, the developer writes the minimum amount of code necessary to pass the test. The focus here is on writing code that fulfills the requirements of the test, nothing more.

Run the Test: The developer runs the test suite to ensure that the newly written code passes all the tests, including the one that was just written.

Refactor Code (if necessary): Once the test passes, the developer may refactor the code to improve its structure, readability, or performance, ensuring that all tests continue to pass.
Repeat: The cycle is repeated for each new piece of functionality, with new tests being written to cover additional cases.

That’s it