**What is TDD (test driven development)**

Test Driven Development (TDD) is a software development methodology where developers write tests for functionality before writing the actual code to implement it.

**The TDD Cycle**

* Red: Write a test for a piece of functionality. The test is expected to fail initially since the feature does not exist yet.
* Green: Write the minimal code required to make the test pass, focusing only on the functionality described by the test.
* Refactor: Improve and optimize the code while ensuring all tests still pass. This step is important for maintaining clean and efficient code.

| **Phase** | **Description** |
| --- | --- |
| Red | Write a failing test first |
| Green | Write code to make the test pass |
| Refactor | Clean up code while ensuring tests pass |

**Key Concepts**

* TDD emphasizes creating automated tests before any feature or function is developed, which helps ensure code correctness from the start.
* It follows an iterative and repetitive cycle known as Red-Green-Refactor

TDD is widely adopted in modern software engineering to improve code quality, reliability, and maintainability

Testing is the process of evaluating software to verify and validate that it performs as expected, is free of defects, and meets user requirements.

**What is Testing?**

* Testing systematically checks the functionality, reliability, and security of software or applications under various conditions.
* It includes both manual and automated methods involving test cases, scripts, or tools to detect errors and confirm correctness.
* The process covers multiple levels, such as unit testing, integration testing, system testing, and acceptance testing

JUnit testing is an open-source testing framework for Java programming, mainly used to create and run automated unit tests to check if individual pieces of code (like methods or classes) function as expected.

**What is JUnit Testing?**

* JUnit allows developers to write small test methods using simple annotations such as @Test to verify the logic of Java code.
* It supports various assertions to compare actual output with expected output, facilitating quick validation of code correctness.
* JUnit can execute hundreds of tests automatically and provides clear pass/fail results, making debugging and development more efficient.

**Why Use JUnit?**

* Automation: JUnit supports running tests automatically, making it easy to check code for errors after every change or during build processes.
* Early Bug Detection: Errors are caught early in development, which reduces the risk of complex bugs or regressions in later stages.
* Code Quality: Consistent use of unit tests with JUnit enhances software reliability and makes the codebase easier to maintain and extend.
* Integration: Works seamlessly with build tools (like Maven and Gradle), IDEs (such as Eclipse and IntelliJ), and CI/CD pipelines, accelerating modern development workflows.
* Regression Testing: Ensures that new code changes do not break existing functionality by enabling repeated, automated testing.

**Key Features**

* Annotations: Mark test cases with keywords like @Test, @Before, @After, etc., to structure and automate the test life cycle.
* Assertions: Methods like assertEquals() and assertTrue() help validate expected outcomes.
* Test Suites: Group and execute sets of related tests efficiently.

JUnit is a foundational tool for Java developers, providing a fast, automated, and reliable way to test and improve code quality throughout the development process

**Architecture to Junit testing**

JUnit testing architecture in its latest version (JUnit 5) is structured into three main components: JUnit Platform, JUnit Jupiter, and JUnit Vintage

| **Component** | **Role** |
| --- | --- |
| JUnit Platform | Core for launching/testing frameworks on JVM |
| JUnit Jupiter | Programming model for JUnit 5 tests |
| JUnit Vintage | Backward compatibility for JUnit 3/4 |

**JUnit Platform**

* Serves as the foundation for running testing frameworks on the Java Virtual Machine (JVM).
* Provides the TestEngine API, letting other frameworks run alongside JUnit and enabling integration with IDEs (Eclipse, IntelliJ), build tools (Maven, Gradle), and CI/CD systems.
* Facilitates test discovery, filtering, and execution through the Launcher interface.

**JUnit Jupiter**

* Contains the programming and extension models for writing modern JUnit 5 tests.
* Includes advanced annotations such as @Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, and extension points to customize test execution.
* Implements its own TestEngine to interpret Jupiter-based tests on the JUnit Platform.

**JUnit Vintage**

* Ensures backward compatibility to run JUnit 3 and JUnit 4 tests on the same platform alongside new JUnit 5 tests.
* Provides a TestEngine for older JUnit test suites, simplifying migration to newer JUnit versions.

| **Aspect** | **TDD (Test Driven Development)** | **BDD (Behavior Driven Development)** |
| --- | --- | --- |
| Focus | Developer-centric: focuses on testing individual code units or functionality correctness | User/business-centric: focuses on system behavior from the end user's perspective |
| Approach | Write tests before writing code; iterative cycle of Red-Green-Refactor to ensure feature correctness | Define behaviors and scenarios in plain language (Given-When-Then) to bridge understanding across stakeholders |
| Language | Uses programming languages for test cases (e.g., Java with JUnit) | Uses natural language or domain-specific language like Gherkin to describe behaviors |
| Collaboration | Mainly developers write and maintain tests | Involves developers, testers, business analysts, and customers collaboratively |
| Test Level | Primarily unit tests targeting small pieces of code | Higher-level tests focusing on system or feature behavior and acceptance criteria |
| Purpose | Ensures code works correctly and is maintainable | Ensures that the software meets business requirements and desired user outcomes |
| Tools | JUnit, NUnit, TestNG, etc. | Cucumber, JBehave, SpecFlow, etc. |

Gherkin is a domain-specific, human-readable language used in Behavior-Driven Development (BDD) to write software behavior test scenarios in a simple, structured format that both technical and non-technical stakeholders can understand.

**What is Gherkin?**

* Gherkin describes software features and scenarios in plain text using a few key keywords.
* It acts as a bridge between business requirements and technical implementation, making collaboration easier.
* Gherkin scenarios can be executed as automated tests using tools like Cucumber

| **Keyword** | **Purpose** |
| --- | --- |
| Feature | High-level description of a software feature or functionality being tested |
| Scenario | A specific test case describing one behavior or example |
| Given | Preconditions or context before an action is performed |
| When | The action or event triggered by the user or system |
| Then | Expected outcome or result following the action |
| And | Used to add additional steps to Given, When, or Then clauses |
| But | Adds a negative or contrasting condition, like “And” but for exceptions |
| Background | Steps common to all scenarios within a feature, run before each scenario |
| Scenario Outline | Template for scenarios that run multiple times with different examples or data sets |
| Examples | A table of values that fills in variables in a Scenario Outline |
| Tags | Labels to organize, filter, and group scenarios |

Example:

Feature: User login

Background:

Given the user is on the login page

Scenario: Successful login with valid credentials

When the user enters valid username "admin" and password "admin123"

Then the user is redirected to the dashboard

Scenario Outline: Login with multiple credentials

When the user enters username "<username>" and password "<password>"

Then the user sees message "<message>"

Examples:

| username | password | message |

| admin | admin123 | Login successful |

| user | wrongpass | Invalid credentials |