LEVEL OF TESTING

Mansi Thakur

Table of Contents

1. Unit Testing....1
   1.1 Types of Unit Testing....1
      A) Positive Testing....1
      B) Negative Testing....1
      C) Boundary Testing....2
      D) Mocking....2
      F) Parameterized Testing....2
2. Integration Testing....2
   2.1 Types of Integration Testing....3
      A) Big Bang Integration Testing....3
      B) Incremental Integration Testing....3
      C) API Testing....3
      D) Stub Testing....3
      F) Driver Testing....3
3. System Testing....3
   3.1 Types of System Testing....4
      A) Functional Testing....4
      B) Non-Functional Testing....4
      C) Regression Testing....4
      D) Stress Testing....4
      E) Recovery Testing....4
4. Acceptance Testing....5
   4.1 Types of Acceptance Testing....5
      A) User Acceptance Testing (UAT):....5
      B) Alpha Testing....5
      C) Beta Testing:....5
      D) Contract Acceptance Testing....6
      E) Regulation Acceptance Testing....6

Testing is an essential part of the software development lifecycle. It ensures that the software behaves as expected, meets business requirements, and is free from defects. There are various **levels of testing** carried out at different stages of development. Each level focuses on different aspects of the application and serves specific purposes. These levels include **Unit Testing**, **Integration Testing**, **System Testing**, and **Acceptance Testing**, among others.

1. Unit Testing

Unit Testing involves testing the smallest testable part of an application, typically a function or method, in isolation from the rest of the system. It ensures that individual pieces of the code work as intended.

A) Why Unit Testing is Important?

- **Early Detection of Errors**: By focusing on individual components, unit testing helps identify bugs early in the development cycle, reducing cost and effort required for fixing later.
- **Improved Code Quality**: Writing unit tests often leads to better-designed code with lower complexity.
- **Refactoring Confidence**: Developers can refactor code with confidence, knowing that unit tests will catch any issues.

B) When to Perform Unit Testing?

- Unit testing is typically done **during the development phase** as the code is written, usually by developers.
- It is the first line of defense in the testing process.

C) How to Perform Unit Testing?

- Write test cases for individual units of code, such as functions, methods, or classes.
- Use **mocking** or **stubbing** to isolate the unit from its dependencies (e.g., databases or external services).
- Focus on testing functionality, edge cases, and error handling.
- Make sure tests are fast and repeatable.

D) Common Unit Testing Frameworks/Tools

- **JUnit** (for Java)
- **NUnit** (.NET)
- **PyTest** (for Python)
- **Mocha** (for JavaScript)
- **Mockito** (for mocking in Java)

1.1 Types of Unit Testing

Unit Testing is the first level of testing, typically focused on individual components or functions. Below are the different types of unit testing:

A) Positive Testing

What: Verifies that the function or unit behaves as expected with valid inputs.
When to Use: When you want to ensure the function returns the correct output for expected, valid inputs.
Example: Testing a function that calculates the sum of two numbers, ensuring that it correctly returns the sum when given valid inputs.

B) Negative Testing

What: Ensures the function handles invalid or unexpected inputs gracefully, without crashing.
When to Use: When you want to ensure the system handles invalid inputs or edge cases.

Example: Testing a function that checks for prime numbers, passing non-integer values or negative numbers to ensure proper error handling.

C) Boundary Testing

What: Verifies that the unit behaves correctly at the edges of input ranges (e.g., minimum, maximum values).
When to Use: To ensure that your system behaves correctly at the boundaries of acceptable input values.
Example: Testing a function that processes a list of integers, ensuring it handles an empty list or a list with only one item correctly.

D) Mocking

What: Involves replacing dependencies of the unit being tested with mock objects that simulate the behavior of real objects.
When to Use: When the unit depends on external systems (like a database or API), which are unavailable or difficult to test directly.
Example: Mocking a database call within a function to isolate testing of the function's logic.

F) Parameterized Testing

What: Uses multiple input values to test a unit's behavior with different data sets, ensuring it works across a wide range of inputs.
When to Use: When the function should behave correctly for different combinations of inputs.
Example: Testing a sorting function with multiple different sets of input arrays.

2. Integration Testing

Integration testing validates the interaction between two or more components or systems. Unlike unit testing, which focuses on individual units, integration testing checks if the units, once integrated, work together as expected.

A) Why Integration Testing is Important?

- **Detects Interface Issues**: This testing ensures that the interfaces between different modules function correctly.
- **Prevents System Failures**: Identifies issues related to data flow, message passing, and shared resources between modules that may not be apparent in unit tests.

B) When to Perform Integration Testing?

- **After unit testing**, once individual modules have been developed and unit tested.
- It is done before system testing, as integration testing focuses on module interaction and system behavior as a whole.

C) How to Perform Integration Testing?

- **Top-Down Approach**: Start testing from the higher-level modules and integrate them progressively, testing each level as you go.
- **Bottom-Up Approach**: Start with testing the lower-level components and move upwards, verifying that the higher-level components are integrated later.
- **Big Bang Approach**: Integrate all components at once and test the overall integration.
- Use **stubs**, **drivers**, and **mock objects** to simulate parts of the system that are not yet developed or are too difficult to integrate in the current phase.
- Verify that data flows correctly, responses are as expected, and communication between modules is seamless.

D) Common Tools for Integration Testing

- **JUnit** and **TestNG** (Java)
- **Postman** (for API integration testing)
- **SoapUI** (for SOAP-based services)

2.1 Types of Integration Testing

Integration Testing focuses on checking the interaction between multiple units or components. Here are the different types:

A) Big Bang Integration Testing

What: Involves integrating all the components at once and testing them together as a whole.
When to Use: When the system is small and components are already stable.
Pros/Cons: Quick to execute, but if an issue arises, it can be difficult to pinpoint the cause.

B) Incremental Integration Testing

What: Components are integrated one at a time and tested as each is added.
When to Use: For larger systems, where individual modules need to be tested progressively.
Types:

1. **Top-Down**: Testing starts with the higher-level modules, and lower-level modules are integrated step-by-step.
2. **Bottom-Up**: Testing starts with lower-level modules, and higher-level modules are added later.
3. **Sandwich**: A combination of top-down and bottom-up approaches, where certain parts of the system are tested first, and others follow in parallel.

C) API Testing

What: Testing the interactions between different services or systems through APIs.
When to Use: When the system relies on APIs to interact with external or internal services.
Example: Testing a payment gateway API integration in an e-commerce system.

D) Stub Testing

What: Using stubs (mock implementations) for lower-level components that are not yet developed or unavailable.
When to Use: During early stages of development when some components are still under construction.
Example: Using a stub for a database service while testing the business logic that interacts with it.

F) Driver Testing

What: Creating a driver to simulate the higher-level module when lower-level components need testing.
When to Use: When the higher-level modules are not yet implemented but need to be tested with lower-level modules.
Example: Using a driver to simulate the user interface while testing the backend logic of a system.

3. System Testing

System Testing is the process of testing the entire system as a complete entity to ensure that all the components, both integrated and individual, work together as expected in the real-world environment.

A) Why System Testing is Important?

- **End-to-End Validation**: It ensures that all modules and integrated systems meet the overall system requirements.
- **Behavior Under Load**: It checks how the entire system functions under different scenarios, loads, and conditions.
- **Comprehensive Test Coverage**: Validates not only functional aspects but also non-functional requirements (e.g., performance, security, and compatibility).

B) When to Perform System Testing?

- **After integration testing** and before acceptance testing.
- Performed by dedicated testers or testing teams, often after the development team completes the system integration.

C) How to Perform System Testing?

- **Functional Testing**: Verify that the system behaves as expected according to functional specifications.
- **Non-Functional Testing**: Perform tests on performance, security, scalability, and reliability.
- **Boundary Testing**: Ensure that the system behaves well with both valid and invalid inputs.
- **Recovery Testing**: Test how the system recovers from failures such as crashes/malfunctions.
- **Stress Testing**: Check the limits of the system under extreme conditions.

D) Common Tools for System Testing

- **Selenium** (for automated UI testing)
- **JUnit** and **TestNG** (for functional testing)
- **JMeter** (for performance and load testing)
- **LoadRunner** (for performance and scalability testing)
- **Appium** (for mobile app testing)

3.1 Types of System Testing

System Testing is the testing of the complete integrated system to verify that it meets the specified requirements. Here are various types of system testing:

A) Functional Testing

What: Verifies that the system's functions work according to the requirements.
When to Use: After integration testing to ensure that all system functionalities are working.
Example: Testing whether a login system works as intended when providing a correct username and password.

B) Non-Functional Testing

What: Focuses on aspects like performance, usability, and security.
Subtypes:

- Performance Testing: Measures how well the system performs under normal or peak load.
- Security Testing: Ensures that the system is protected against unauthorized access and vulnerabilities.
- Usability Testing: Evaluates the user interface and user experience.
- Compatibility Testing: Ensures that the system works across different platforms, browsers, devices, etc.

When to Use: Once functional requirements are validated, to ensure the system works under various conditions and meets all quality attributes.

C) Regression Testing

What: Ensures that changes to the system have not introduced new defects into the previously working functionality.
When to Use: After bug fixes, updates, or new features have been added.
Example: Re-running existing test cases to ensure that recent updates have not broken existing features.

D) Stress Testing

What: Evaluates how the system behaves under extreme conditions (e.g., high load, low resources).
When to Use: To identify breaking points or performance bottlenecks.
Example: Simulating thousands of simultaneous users on a web application.

E) Recovery Testing

What: Tests the system's ability to recover from hardware or software failures.
When to Use: After system setup or updates.
Example: Disconnecting a database to ensure the system can reconnect automatically.

4. Acceptance Testing

Acceptance testing is the final level of testing where the system is tested against the business requirements to determine whether it is ready for deployment. This can be performed by the customer, end-users, or a QA team in collaboration with stakeholders.

A) Why Acceptance Testing is Important?

- **Business Validation**: Ensures that the system meets business goals and expectations.
- **User-Focused**: Tests the system's readiness from a user perspective.
- **Final Checkpoint**: Verifies that the system is ready for production deployment.

B) When to Perform Acceptance Testing?

- Acceptance testing is carried out **after system testing** when the system is fully integrated and validated from a technical perspective.
- It is typically the **last phase of testing** before the software is released to production.

C) How to Perform Acceptance Testing?

- **User Acceptance Testing (UAT)**: End-users perform tests based on business requirements and scenarios to verify if the system meets their expectations.
- **Alpha Testing**: In-house testing conducted by the development team before releasing to external users.
- **Beta Testing**: Performed by a select group of external users who provide feedback before the official release.
- **Test Criteria**: Prepare test cases based on user stories, use cases, and business requirements.

D) Common Tools for Acceptance Testing

- **TestRail** (for managing and tracking test cases)
- **JIRA** (for issue tracking and project management)
- **Cucumber** (for Behavior-Driven Development, involving business stakeholders in testing)
- **Selenium** (for automating UAT scenarios)

4.1 Types of Acceptance Testing

Acceptance Testing is the final stage of testing before a product is deployed into production. It ensures the system meets business needs. Below are the key types of acceptance testing:

A) User Acceptance Testing (UAT):

What: Verifies that the software works as intended for the end-user and meets business requirements.
When to Use: Typically performed by business users or clients to verify that the system meets their expectations.
Example: A business user tests an e-commerce website to confirm that all purchasing features work as expected.

B) Alpha Testing

What: Performed by the development team in a controlled environment before being released to a limited group of external testers.
When to Use: After the internal development phase but before beta testing.
Example: Developers test a new app release internally to identify bugs and refine features.

C) Beta Testing:

What: Performed by a limited number of external users to gather feedback before the final release.
When to Use: Before releasing the product to the entire user base.
Example: Releasing a new version of a mobile app to a selected group of users for feedback.

D) Contract Acceptance Testing

What: Testing that ensures the product meets the agreed-upon terms outlined in a contract.
When to Use: When the software is delivered to the client based on a signed contract.
Example: Testing an application based on the contract's deliverables to ensure the client's specific requirements are met.

E) Regulation Acceptance Testing

What: Verifies that the system complies with legal, regulatory, or compliance standards.
When to Use: For systems subject to industry-specific regulatory standards (e.g., healthcare, finance).
Example: Testing a healthcare application to ensure it meets HIPAA regulations for patient privacy.