**INTRODUCTION TO TESTING METHODOLOGIES**

Research and understand the concepts of unit testing and integration testing in software development.

**TESTING**

When each module of the system has been created it must be tested to ensure that it works correctly. Example of modules that should be tested include:

* Data structures - do data tables have the correct data correctly.
* Validation rules – does the system reject incorrect data.
* Input screens – does each form control allow users to enter data correctly.
* Output screens – are the output results clear.

Testing the individual’s modules of the system is known as unit testing. Any errors will be corrected by the programmer. The module that failed will require it to be re-tested again to make sure the error is fixed.

**TEST PLAN**

Testing a system involves creating and using a test plan. A system plan should be created for each module and should list the different tests that are going to be performed. A good test plan should include:

* List of tests to be performed.
* The data to be used in testing.
* The type of test.
* The expected outcome.
* The actual outcome.

A test plan should always use four types of data:

* Normal data. Accepted data.
* Extreme data. Accepted data in the border of what the system will accept.
* Abnormal data. Unaccepted data.
* Live data. Actual data used after the installation of the system.

**SOFTWARE TESTING**

Software testing can be categorized into white-box and black-box testing.

**BLACK BOX TESTING**

It is a software testing technique in which the internal structure of the system is not known by the tester. Only the external design and structure are tested. The tester only focuses on input and the output of the software.

**WHITE BOX TESTING**

It is a software testing technique in which the internal structure of the system is not known by the tester. Implementation and impact of the code are tested. The tester has knowledge of the internal working of the software and can test the individual code snippet.

**TEST PLAN EXAMPLE**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TEST NAME | TEST DATA | EXPECTED RESULT | ACTUAL RESULT | COMMENTS | TEST TYPE |
| User Login | Correct User ID or Credentials | Should Login | Login accepted | Confirmed | Black box test. Normal data |
| User Login | Incorrect User credentials | Should not login | Login unaccepted | Error message | Black box test.  Abnormal data |
| Register User | New User ID registered | Should register user | Register accepted | Confirmed | White Box.  Normal data |
| Register User | Existing User ID | Should not register user | Register unaccepted | Error Message | White Box. Extreme data |

**UNIT TESTING**

Unit testing involves testing individual components or units of code that is functions, methods or classes to ensure that they work as expected. Unit testing is meant to isolate each part of the system and verify that the logic is correct, without depending on other parts of the system.

**KEY CONCEPTS OF UNIT TESTING**

**Testing in Isolation**

Each unit test should focus on testing one function, method or class in complete isolation from other parts of the system. External dependencies like databases, APIs, and file systems should not be involved in unit tests. If necessary, these dependencies can be replaced with mock objects.

**Test-Driven Development (TDD)**

This is a software development approach where developers write unit tests before writing the actual code. The process follows a Red-Green-Refactor cycle, **Red** write a failing unit test, **Green** write just enough code to pass the test, **Refactor** improve the code without changing its behavior, ensuring that the test still passes. This methodology ensures that tests guide the design and implementation of the code.

**Mocking and Stubbing**

**Mocks** - Mocks simulate the behavior of complex objects (like databases or services) to isolate the unit under test. They help remove dependencies that might slow down or complicate the tests. **Stubs** - Stubs are similar to mocks but return predefined responses without performing any actual computation or accessing real resources. E.g. If your function calls an external API, you can use a mock object to simulate the API call, ensuring that the unit test is independent of external systems.

**Code Coverage**

**Code coverage** is a metric used to measure the percentage of code that is exercised by unit tests. Higher code coverage indicates that a greater portion of the codebase is tested, but it doesn’t guarantee that all edge cases are covered. Coverage tools track how many lines, branches and functions are covered by tests. It’s important to aim for good coverage, but it shouldn’t be the only indicator of test quality.

**CHARACTERISTICS**

**Isolation** - Each test should focus on a single unit of code, without external dependencies like databases, APIs or other modules.

**Fast Execution** - Since unit tests focus on specific pieces of functionality, they run quickly, allowing developers to test frequently without significantly impacting development time.

**Frequent Execution and Automation** - Developers typically run unit tests during development to catch issues early. Unit tests are typically automated, meaning they can be run by test runners without manual intervention. This makes it easier to test consistently and frequently, especially in Continuous Integration environments.

**Independent -** Each unit test should be independent of other tests. This ensures that a failure in one test doesn't affect the outcome of others, making it easier to diagnose issues.

**Focused on Small Parts** - Unit tests are focused on testing individual units that is small, specific parts of the code, such as a single function or method, without considering how they interact with other parts of the system.

**BENEFITS**

**Early Bug Detection** - Unit testing allows developers to catch bugs early in the development process. Since each unit is tested individually, issues can be detected before they propagate to other parts of the application, this reduces the cost and effort of fixing issues later in the development cycle.

**Improved Code Quality** - By thoroughly testing each unit, unit testing ensures that the code performs as expected in various scenarios. This improves the overall reliability and robustness of the application.

**Simplified Refactoring** - Having a suite of unit tests provides a safety net when refactoring code. Developers can confidently change or improve the codebase, knowing that any issues will be caught by the tests.

**Documentation** - Unit tests act as living documentation for the codebase. Each test case shows how a function or module is expected to behave, making it easier for new developers to understand the code.

**Faster Debugging** - Since unit tests focus on small, isolated parts of the code, it becomes easier to pinpoint the exact location of a bug when a test fails. This speeds up the debugging process.

**Increases Confidence During Deployment** - Having comprehensive unit tests ensures that the code functions correctly, which boosts developer confidence when deploying new changes or updates.

**SCENARIO 1**

When building a function to calculate the total of expenses in a month, unit testing ensures that the function returns the correct result for various input scenarios such as empty arrays, arrays with one expense, or arrays with multiple expenses.

**INTEGRATION TESTING**

Integration testing focuses on testing how different components or modules of an application work together. It ensures that various parts of a system interact correctly, verifying that the units integrate without issues.

**TYPES OF INTEGRATION TESTING:**

Big Bang - All units are tested together after integration.

Top-Down - Testing from higher-level modules to lower-level modules.

Bottom-Up - Testing starts with lower-level modules, then progresses upwards.

**KEY CONCEPTS OF INTEGRATION TESTING**

**Testing Interactions Between Components**

Focus on Interfaces and Communication **-** Integration testing ensures that different components or modules communicate and interact with each other properly. It tests the interfaces between them to verify that data flows correctly and the components work in harmony. Example: Testing if a user registration form sends data to the backend and correctly saves it in the database.

**Real-World Scenarios**

Integration testing mimics real-world scenarios by ensuring that multiple components work together as they would in production. It simulates real system behavior, validating how different parts of the system collaborate to achieve a specific function. E.g. An e-commerce site where a user adds an item to the cart, proceeds to checkout, and completes a payment involves the integration of several systems like user authentication, database interaction, and payment gateway.

**Testing Against External Systems**

Unlike unit tests, integration tests often involve external systems like databases, APIs, third-party services, file systems and network connections. These external dependencies must be tested to ensure the system works properly in the presence of these resources. E.g. An API that pulls data from a third-party service and displays it on the application requires integration testing to ensure it works with the external service.

**Detecting Interface Issues**

One of the main goals of integration testing is to detect issues that occur at the boundaries between different components. These are usually interface-level problems such as incorrect data formatting, mismatched function signatures, or communication breakdowns. E.g. Testing whether the frontend correctly sends form data in the expected format to the backend API.

**Slower Than Unit Tests**

Involvement of multiple systems, Since integration tests involve testing interactions between different components and often involve external systems they are generally slower than unit tests. Each component has to be initialized and external systems e.g. databases or APIs may take time to respond.

**Greater Complexity**

Testing Entire Workflows, testing is more complex than unit testing because it involves multiple components interacting with each other. Tests must consider the broader system and workflows rather than focusing on small, isolated units. Example: Testing a login system involves integrating the user interface, authentication service, database, and session management.

**CHARACTERISTICS**

**Interdependence/Tests Multiple Units Together** - Unlike unit testing, which focuses on isolated components, integration testing checks how different units/modules work together. These tests verify that the interactions between components are functioning correctly.

**Involves External Dependencies** - Integration tests often involve components that interact with external systems like databases, APIs, and file systems. The goal is to ensure the correct flow of data and communication between these units.

**Real-World Scenarios** - Integration tests simulate more realistic scenarios where multiple modules work together. For example, they could verify that a user can submit a form that updates a database and triggers other system functions.

**Covers End-to-End Functionality** - Integration tests aim to check that an entire process works as expected. They ensure that multiple components integrate properly and cover the broader functionality of the system.

**Requires More Setup** - Since multiple components interact, setting up integration tests often involves configuring databases, APIs, or services. This setup can be more involved compared to the isolated nature of unit tests.

**BENEFITS**

**Ensures Components Work Together -** Integration testing verifies that individual components interact correctly with each other. This ensures that the system functions as expected when units are integrated, catching issues related to communication between components.

**Detects Interface Defects** Many bugs occur at the interfaces where two components communicate. Integration testing identifies problems like incorrect data passing, broken API contracts, or mismatched function signatures.

**Catches Issues Missed by Unit Tests** - Unit tests focus on testing components in isolation, which means integration issues like incorrect data handling between modules are often missed. Integration testing catches these issues early in the process.

**Prevents System Level Failures** - By ensuring that all components integrate and work as a system, integration testing reduces the likelihood of failures once the application is deployed. It ensures the robustness of the application in complex environments.

**Supports Incremental Development** - Integration testing can be done incrementally as new modules are integrated into the system. It ensures that newly added components don’t break existing functionality and the overall system continues to work as expected.

**Confidence in Deployment** - When integration tests pass, it gives the team confidence that the application is functioning as intended in various environments, reducing the risks associated with deploying new changes or updates.

**EXPLORE THE IMPORTANCE OF TESTING IN ENSURING THE FUNCTIONALITY, RELIABILITY, AND PERFORMANCE OF APPLICATIONS.**

**1.**

**The Importance of Testing in Ensuring Functionality**

Functionality refers to the ability of the application to perform its intended tasks according to the requirements.

**Verification against requirements** - Functional testing checks whether the software meets the functional requirements specified during the design phase. It ensures that all features work as expected and each module behaves according to its specifications. E.g. In an e-commerce application, functional testing would verify that users can add items to a cart, complete a purchase, and receive an order confirmation.

**Unit testing** - Unit testing verifies that individual components or functions of the application behave as intended. This helps detect bugs early in the development process. E.g. Testing a login function to ensure it properly authenticates users with valid credentials and rejects invalid ones.

**Integration Testing** - Integration tests ensure that different components of the application work together seamlessly. It verifies that the interactions between modules (frontend and backend) work as expected.E.g. Testing the integration between a payment gateway and the order management system in an e-commerce platform to ensure payments are processed correctly.

**User Interface (UI) Testing** - UI testing ensures that the graphical user interface behaves as expected and that all buttons, forms, and links function correctly. Example: Testing whether clicking on a submit button properly sends form data to the server and shows a confirmation message.

**2.**

**The Importance of Testing in Ensuring Reliability**

Reliability refers to the consistency of the application’s performance over time and under varying conditions. A reliable application should function without failure during its expected usage.

**How Testing Ensures Reliability**

Regression Testing - Regression testing checks whether recent changes or updates to the codebase have broken existing functionality. It ensures that the system continues to function correctly after new features, bug fixes or enhancements are introduced. E.g. After adding a new feature, regression tests ensure that other core functionalities still work.

**Automated Testing** - Automated testing allows for frequent and consistent testing across various parts of the system, which helps maintain the reliability of the application over time.E.g. Running a suite of automated tests every time new code is pushed ensures that the system remains stable.

**Stability Testing** - Stability testing involves running the application under normal operating conditions for an extended period to ensure it does not crash or degrade in performance. E.g. Running a messaging app continuously for 24 hours to verify that it remains functional and doesn’t encounter memory leaks or crashes.

**Error Handling -** Tests verify that the system handles unexpected inputs or errors gracefully. Error handling ensures that when things go wrong such as network failures or incorrect inputs, the system provides informative error messages and remains stable.E.g. Testing a web form to ensure that invalid email formats trigger appropriate error messages and do not cause the application to crash.

**3.**

**The Importance of Testing in Ensuring Performance**

Performance testing evaluates how well an application behaves under specific conditions, such as high loads or limited resources. It ensures that the application is fast, scalable and can handle high demand without performance degradation.

**Load Testing** - Load testing evaluates the system’s ability to handle a specified load of users, transactions or data volume over a certain period. It simulates real-world usage scenarios such as multiple users accessing the system simultaneously.

**Stress Testing** - Stress testing pushes the system beyond its capacity limits to evaluate its behavior under extreme conditions. This helps identify breaking points and determine how the system fails, gracefully or catastrophically. E.g. Overloading a video streaming service with concurrent requests to check how the server handles the extra load and whether it can recover after failure.

**Response Time Testing** - Response time testing measures how quickly the application responds to user actions or requests. It ensures that performance remains within acceptable limits, even under high loads. E.g. Measuring how long it takes for an online store’s product search feature to return results under normal and peak traffic.