Software Quality Assurance

Lecture 9

Outline

- Testing?
- Defects?
- Manual vs automation
- Baselines
- Test cases
- Test suites
- Test coverage and requirement coverage
- Test driven development
- Whitebox vs blackbox testing
- Types of testing

- 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.”

Testing vs QA

- Testing is a subset of QA.
- While testing is primarily concerned with identifying defects in the software product, QA is concerned with ensuring that the entire software development process produces a high-quality product.
- Testing is a reactive process, whereas QA is proactive, aiming to prevent defects from occurring in the first place.
- Both testing and QA are crucial for delivering high-quality software products to end-users.

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.

Manual vs automated/automation testing

- Manual testing verifies a software product against functional and non-functional requirements manually executed by the QA team.
- QA team will run the product under test on different test environments, precisely as the end users would, to find any deviations from the original software requirements before software deployment.
- In this process, the team will execute the test cases for checking all the features and possible user scenarios and generate the test reports without the help of any automation software testing tool.

- Automation testing is the process of writing code/test scripts to automate the test execution faster, cheaper, and more efficiently.
- Appropriate automation tools to develop the test scripts and validate the software are used for comparing the actual results against the expected results.
- This means that the QA team will write code or scripts on different tools to run the test cases automatically.
- This will help the team to determine whether or not the application performs as expected. Even though all processes are performed automatically, automation requires some manual effort to create initial testing scripts.

- In manual testing, a human performs the tests step by step, without test scripts. In automated testing, tests are executed automatically via test automation frameworks, along with other tools and software.
- The biggest difference between manual and automation testing is who executes the test case. In manual testing, the human tester does it. In automation testing, the tool does it.

When to use Manual testing and when to use Automation testing?

	Benefits	Limitations
Manual	freedom to adapt to requirement changes instantly	Human error Expensive for large projects
Automated	Speed Re-usability Immediate results	Complex maintenance if requirement changes frequently Requires time to design tests Check only predefined issues

The right mix between automation and manual tests is essential for a successful QA team. It might also be suitable for a tester to switch between manual and automation testing from time to time.

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 **reference point** for comparison to track changes and ensure consistency throughout the development and testing process.
- Baselines help in **managing changes** 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.

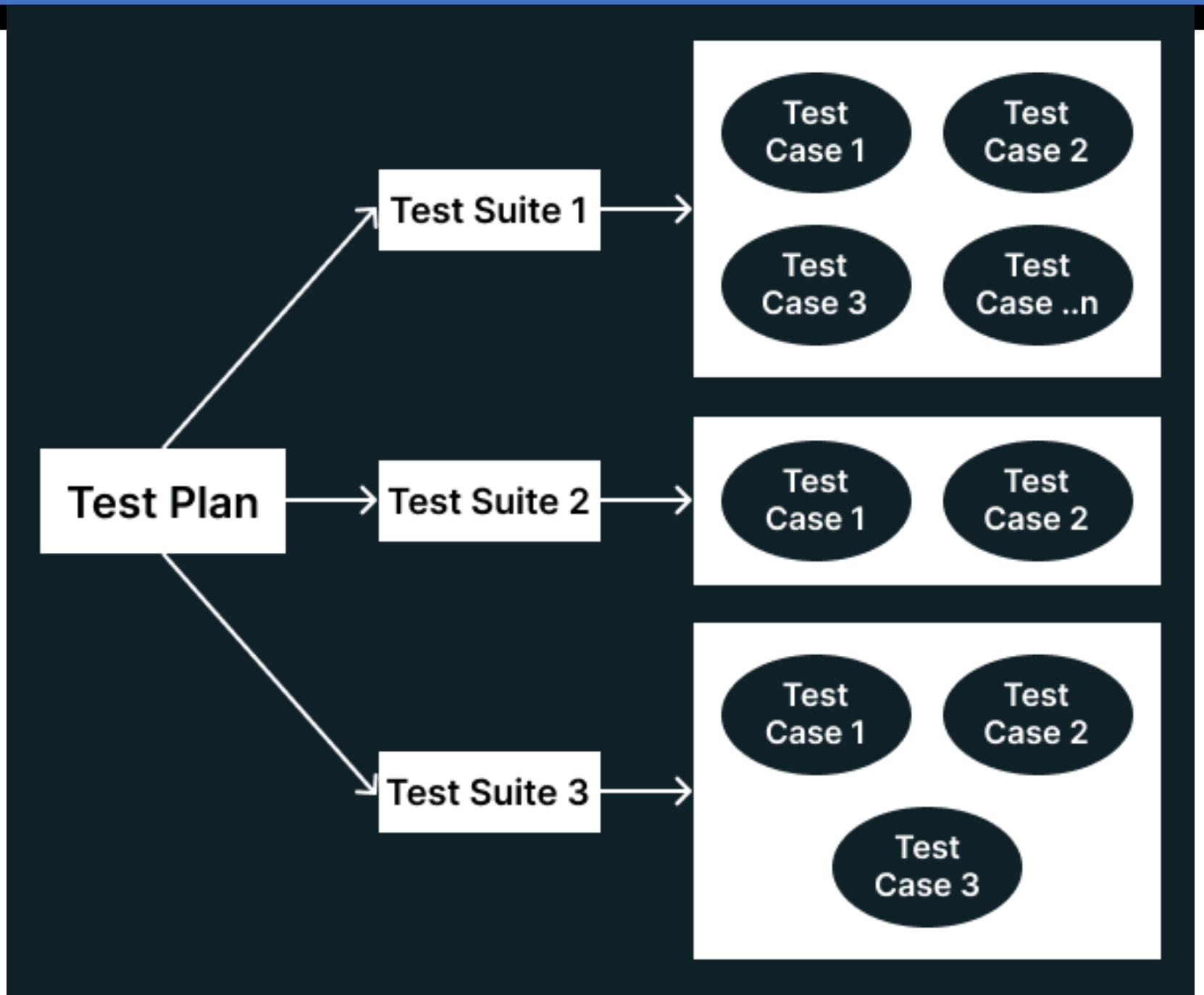
Test suites

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.

Whitebox testing vs blackbox testing

White-box testing

- White-box testing, also known as clear-box testing, glass-box testing, or structural testing
- White-box testing aims to validate the correctness of the internal workings of the software, including the control flow, data flow, conditional statements, loops, and error-handling paths.
- It ensures that all statements, branches, and paths within the code are executed and tested thoroughly, helping to identify defects, errors, and vulnerabilities in the codebase.

Testing Techniques:

- White-box testing techniques include statement coverage, branch coverage, path coverage, condition coverage, and loop coverage, among others.
- Test cases are designed to exercise specific code paths and conditions within the software to verify that they behave as expected and handle all possible scenarios.

Types of White-box Testing:

- **Unit Testing**: In unit testing, individual units or components of the software, such as functions, methods, or classes, are tested in isolation from the rest of the system. White-box techniques, such as code coverage analysis, are commonly used to ensure thorough testing of the codebase.
- **Integration Testing**: White-box techniques are also applied in integration testing to verify the interactions and interfaces between integrated components or modules of the software. This may involve testing APIs, data flows, and communication channels between subsystems.

Black-box testing

- Black-box testing, also known as behavioral testing or functional testing, focuses on testing the functionality of the software without considering its internal structure or implementation details.
- Testers approach the system as a "black box," where they are unaware of its internal workings and only interact with it based on its specifications and inputs.
- Test cases are derived from requirements specifications, use cases, user stories, or other external documentation without knowledge of the internal code implementation

Test Techniques: Various techniques can be employed in black-box testing, including equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and exploratory testing.
Testing Levels: Black-box testing can be applied at different levels of testing, including unit testing, integration testing, system testing, and acceptance testing.

Types of testing

Unit Testing:

- Unit testing is a software testing technique where individual units or components of a software application are tested in isolation.
- A unit refers to the smallest testable part of an application, typically a function, method, or class.
- The purpose of unit testing is to validate that each unit of the software performs as expected, according to its design and specifications.

Let's consider a scenario of building a basic banking system. In this system, we can identify several units, such as account management, transaction processing, authentication, and reporting.

1. **Account Management Unit**: This unit handles tasks related to managing bank accounts, including creating new accounts, updating account information, and deleting accounts.
2. **Transaction Processing Unit**: This unit is responsible for processing various types of transactions, such as deposits, withdrawals, transfers between accounts, and balance inquiries.
3. **Authentication Unit**: This unit manages user authentication and authorization processes. It ensures that only authorized users can access their accounts and perform transactions.
4. **Reporting Unit**: This unit generates various types of reports, such as account statements, transaction histories, and summaries of account balances.

Integration Testing:

Integration testing involves testing the interactions and interfaces between different components or modules of the software to ensure they work together as expected.

Let's consider a scenario where we have a distributed system for a ride-sharing service. In this system, various components interact with each other, such as the user interface (mobile app), the ride request dispatcher, the driver allocation service, and the payment processing system.

Components:

1.Mobile App (User Interface): This component represents the user interface where users can request rides, view available drivers, and track their ride status.

2.Ride Request Dispatcher: This component receives ride requests from users and dispatches them to available drivers based on factors like distance, driver availability, and user preferences.

3.Driver Allocation Service: This component manages the allocation of drivers to ride requests, considering factors like driver location, current ride status, and driver availability.

4.Payment Processing System: This component handles payment transactions for completed rides, charging users' credit cards and disbursing payments to drivers.

- The Ride Request Dispatcher receives ride requests from users and interacts with the Driver Allocation Service to assign drivers to these requests.
- The Driver Allocation Service manages the list of available drivers and allocates them to ride requests based on factors such as their current availability and proximity to the pickup location.

Integration Testing:

- To perform integration testing, we would simulate a ride request being dispatched by the Ride Request Dispatcher and verify that the Driver Allocation Service correctly assigns a driver to the request. This ensures that the interaction between these two components works as expected and that drivers are allocated efficiently based on the given criteria.

Regression Testing

Regression testing involves re-running previously executed tests to ensure that recent code changes have not introduced new bugs or caused existing functionality to regress.

Scenario:

Initial Testing: The initial version of the application is tested thoroughly, covering functionalities like user registration, product search, adding items to the cart, and checkout process. All tests pass successfully.

New Feature Implementation: The development team adds a new feature that allows users to filter products by price range. They implement this feature and perform testing specifically related to this new functionality.

Regression Testing:

1. After the new feature is implemented, the testing team performs regression testing to ensure that the existing functionalities (like user registration, product search, adding items to the cart, and checkout process) are still working as expected.
2. They run a set of predefined test cases that cover the core functionalities of the application.
3. These test cases include scenarios such as:
   - ✓ Registering a new user
   - ✓ Logging in with existing credentials
   - ✓ Searching for products by name, category, and now by price range
   - ✓ Adding items to the cart
   - ✓ Proceeding to checkout and completing the purchase

Smoke testing

Smoke testing, also known as build verification testing or sanity testing, is a type of software testing that focuses on quickly determining whether the most critical functionalities of an application work properly. It is typically performed early in the testing process, often after a new build or deployment, to ensure that the application is stable enough for more comprehensive testing.

Scenario: You're part of a software development team working on an e-commerce website. After completing a round of bug fixes and enhancements, the team is ready to deploy the latest version of the website to the testing environment. Before proceeding with extensive testing, the team decides to perform smoke testing to quickly verify that the critical functionalities are working as expected.

Verify Basic Functionality: The tester navigates to the homepage of the e-commerce website and ensures that it loads without any errors. They check that essential elements such as the navigation menu, search bar, and footer are present and functioning correctly.

Test User Authentication: The tester attempts to log in using valid credentials and verifies that they can access their account dashboard without encountering any authentication-related issues. They also check the functionality of the "Forgot Password" feature to ensure that users can reset their passwords if needed.

Test Product Browsing: The tester browses through different product categories, adds items to the shopping cart, and verifies that they can proceed to checkout without encountering any errors. They also check that product images, descriptions, and prices are displayed correctly.

Test Checkout Process: The tester goes through the checkout process, entering shipping and payment information, and completes a test order. They verify that the order is processed successfully, and they receive a confirmation message or email with the order details.

Test Key Functionality: The tester performs quick checks on other critical functionalities, such as adding items to a wishlist, applying discount codes, and viewing order history. They ensure that these features are functioning as expected and do not show any unexpected behavior.

Document Results: The tester records the outcomes of the smoke test, noting any issues or anomalies encountered during the testing process. If the smoke test passes without any critical issues, the team can proceed with more extensive testing. Otherwise, they may need to investigate and address the identified issues before continuing.

Regular testing vs smoke testing

Regular testing covers a wide range of test scenarios, including positive and negative test cases, edge cases, boundary conditions, and user interactions, to ensure thorough validation of the software's features and functionalities.

Smoke testing focuses on a subset of essential functionalities and critical paths of the application, typically covering high-priority features and core workflows that are crucial for the application's basic functionality and usability.

Regular testing is typically performed after smoke testing and focuses on more extensive testing efforts, including regression testing, integration testing, system testing, acceptance testing, and other types of testing as needed throughout the software development lifecycle.

Smoke testing is performed early in the testing process, often immediately after a new build or deployment, to quickly assess the stability and readiness of the application for more extensive testing efforts.

Regular testing requires more time and resources compared to smoke testing due to its comprehensive nature and the need to cover a wide range of test scenarios and conditions.

Smoke testing is designed to be fast and lightweight, focusing on rapid verification of critical functionalities within a short time frame, allowing testers to quickly identify any major issues or regressions introduced in the latest build.

Acceptance testing

Acceptance testing is a type of software testing that focuses on evaluating whether a system meets the requirements and expectations of its end-users or stakeholders.

Types:

- **User Acceptance Testing (UAT)**: UAT involves testing the software in a real-world environment by actual users or representatives of the end-users. It focuses on validating whether the software meets the user's needs, preferences, and business requirements.
- **Business Acceptance Testing (BAT)**: BAT focuses on verifying whether the software aligns with the business objectives and requirements defined by the organization. It ensures that the software supports the business processes and goals effectively.
- **Operational Acceptance Testing (OAT)**: OAT focuses on evaluating whether the software can be deployed and operated in the production environment. It includes testing aspects such as installation, configuration, performance, scalability, and reliability.

System testing

System testing is conducted after integration testing and before acceptance testing, typically in a testing environment that closely resembles the production environment.

Functional Testing: System testing validates that the system functions correctly according to its functional requirements. It includes testing various use cases, scenarios, and workflows to ensure that the system performs its intended tasks accurately.

Non-Functional Testing: System testing also evaluates non-functional aspects such as performance, scalability, reliability, security, and usability. This may involve stress testing, load testing, security testing, usability testing, and other types of testing to assess the system's behavior under different conditions.

System testing is typically performed in a dedicated testing environment that closely resembles the production environment, including hardware, software, networks, and configurations.

The test environment may include simulated or representative data to mimic real-world usage scenarios and conditions.

That’s it