MODULE 6

SOFTWARE TESTING & MAINTENANCE

Tarunima Mukherjee

Software Testing

Testing is the process of executing a program with the aim of finding errors. To make our software perform well it should be error-free. If testing is done successfully it will remove all the errors from the software.

Principles of Testing: -

1. All the test should meet the customer requirements

2. To make our software testing should be performed by a third party

3. Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.

4. All the test to be conducted should be planned before implementing it

5. It follows the Pareto rule (80/20 rule) which states that 80% of errors come from 20% of program components.

6. Start testing with small parts and extend it to large parts.

Software Testing

Why to Learn Software Testing?

• In the IT industry, large companies have a team with responsibilities to evaluate the developed software in context of the given requirements. Moreover, developers also conduct testing which is called **Unit Testing**.

• In most cases, the following professionals are involved in testing a system within their respective capacities −

- ❑ Software Tester
- ❑ Software Developer
- ❑ Project Lead/Manager
- ❑ End User

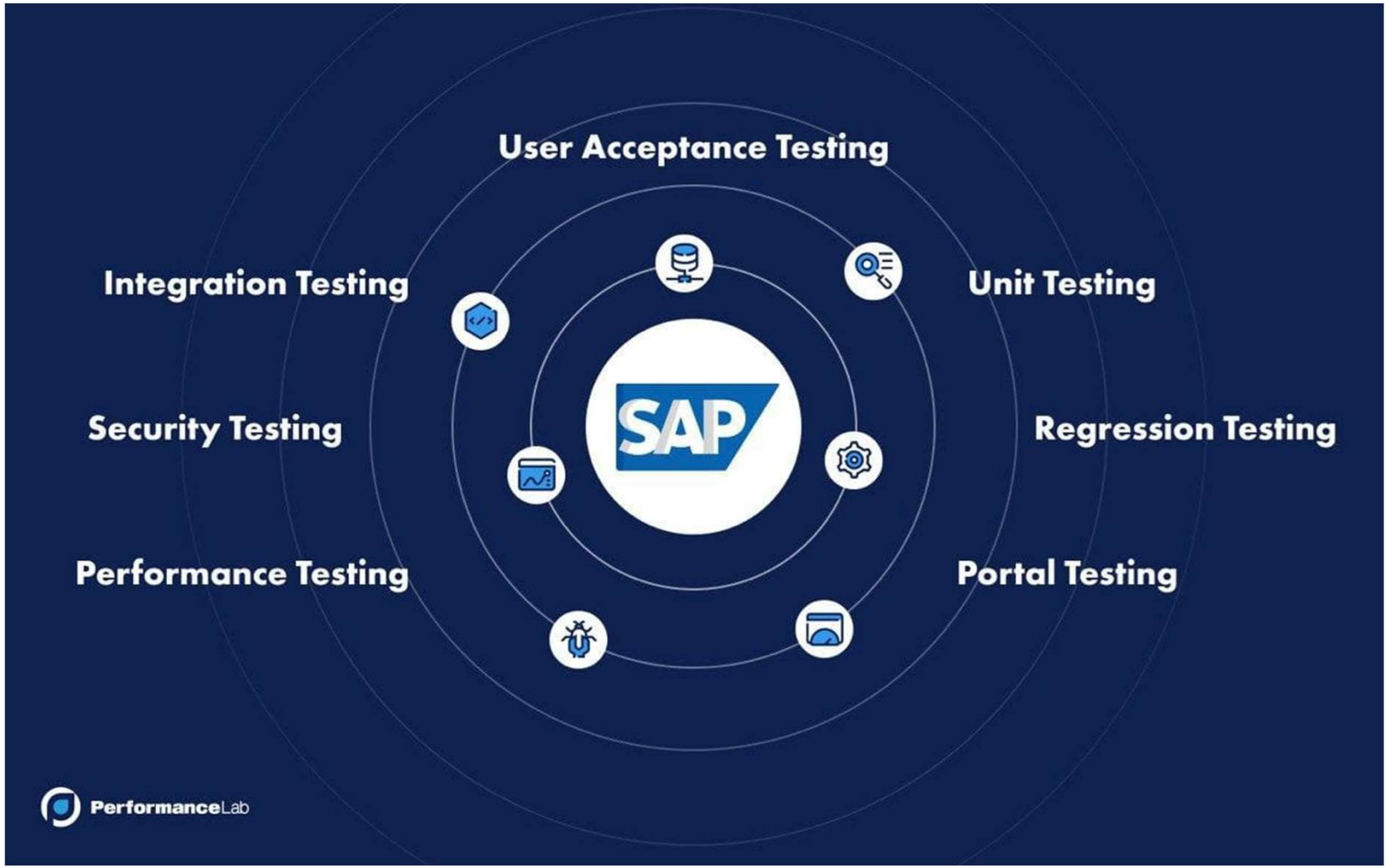
Software Testing

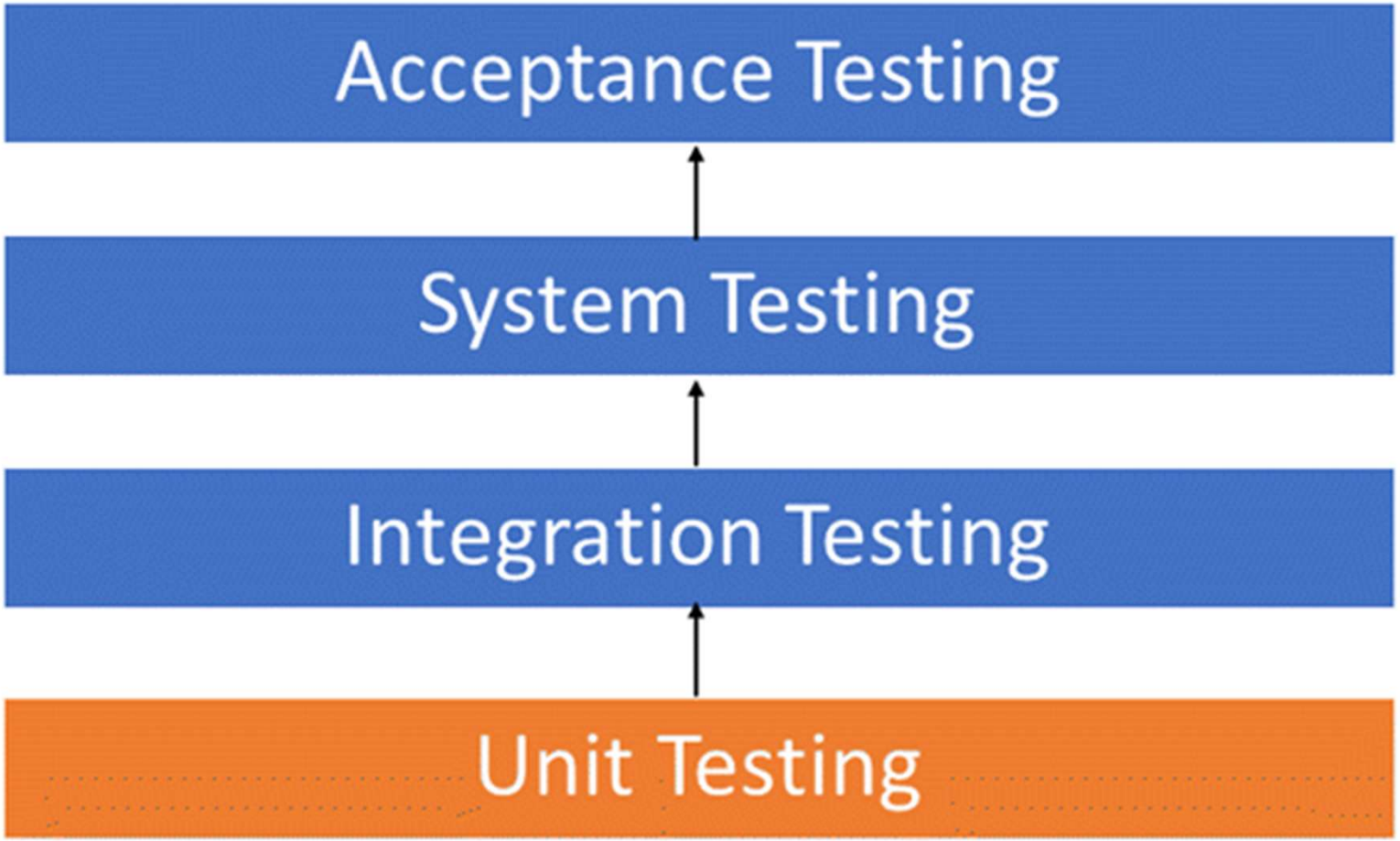
SOFTWARE APPLICATION TESTING SERVICES

The Software Testing Team at Watzmann provide the below mentioned Application Testing Services:

- **Enterprise Application Testing** - Enterprise applications are critical to many businesses across various industries. At Watzmann Consulting we have a proficient QA team that can support infrastructure and processes to optimize testing of business applications.
- **Web Application Testing** - Watzmann's dedicated team of testing engineers have good experience and knowledge in web application testing technologies. We are proficient in different technologies has generated in delivering multiple projects at different parts of the globe.
- **Desktop Application Testing** – We provide a wide range of testing services on desktop applications. With the support of our detailed desktop application testing expertise and QA testing experience we have gained substantial experience in this.
- **Mobile App Testing** - Our mobile app testing services team has knowledge of handling complex apps which are intended to function across multiple devices with varying screen sizes, internal hardware, resolutions that can operate under various operating systems.


Standard / General Testing
Integration Testing
System Testing
Acceptance Testing
Usability Testing
Sanity Testing
Functional Testing
Compatibility Testing
Accessibility Testing
Domain-Based Testing Services
Health Care Testing
Game Testing
Mobile Testing
CMS Testing
CRM Testing
Publishing Testing
Specialized Testing Services
Enterprise App
Mac Testing
Web Application
Data Base Testing
Desktop App
Automated Testing Services
Regression Testing
Ad-Hoc Testing
Data-Driven
Model Driven
Load Testing
Stress Testing
Performance Testing


User Acceptance Testing
Integration Testing
Unit Testing
Security Testing
SAP
Regression Testing
Performance Testing
Portal Testing
PerformanceLab


Acceptance Testing
System Testing
Integration Testing
Unit Testing

Unit Testing

UNIT TESTING is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

Why Unit Testing?

Unit Testing is important because software developers sometimes try saving time doing minimal unit testing and this is myth because inappropriate unit testing leads to high cost Defect fixing during System Testing, Integration Testing and even Beta Testing after application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Unit Testing is of two types:

- Manual
- Automated

Unit testing is commonly automated but may still be performed manually. Software Engineering does not favor one over the other but automation is preferred. A manual approach to unit testing may employ a step-by-step instructional document.

Unit Testing Techniques

The **Unit Testing Techniques** are mainly categorized into three parts which are Black box testing that involves testing of user interface along with input and output, White box testing that involves testing the functional behaviour of the software application and Gray box testing that is used to execute test suites, test methods, test cases and performing risk analysis.
Code coverage techniques used in Unit Testing are listed below:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Finite State Machine Coverage

Unit tests are run automatically on the software engineers' machines during the development process. By using intelligent tools, only the tests affected by the changes made to the relevant software module are run, rather than the whole test suite.

Unit Test Example: Mock Objects

Unit testing relies on mock objects being created to test sections of code that are not yet part of a complete application. Mock objects fill in for the missing parts of the program.
For example, you might have a function that needs variables or objects that are not created yet. In unit testing, those will be accounted for in the form of mock objects created solely for the purpose of the unit testing done on that section of code.

Integration Testing

What Is Integration Testing?

- Let's start with a formal definition. Integration testing is a type of testing meant to check the combinations of different units, their interactions, the way subsystems unite into one common system, and code compliance with the requirements.
- For example, when we check login and sign up features in an e-commerce app, we view them as separate units. If we check the ability to log in or sign up after a user adds items to their basket and wants to proceed to the checkout, we check the integration between these two functionalities.

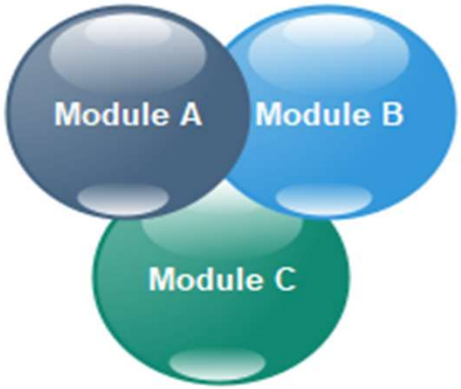
Why Do We Need Integration Testing?

- IT specialists are well aware of how unpredictable code can be, even if it seems perfectly written. Besides, there are dynamic projects, where pre-approved requirements can lose their relevance and change – that's especially true for Agile projects.
- Simultaneous testing of different modules is very convenient, practical, and cost-efficient.
- Integration checks can take place at any stage of the SDLC.
- It is a lifesaver for a team involved in projects with constantly changing requirements or logic that is under review after development has started.
- Unlike other types of tests, integration can cover any amount of program code in one sprint.

Similarly Mail Box: Check its integration to the Delete Mails Module.

Test Case ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mailbox select the email and click a delete button	Selected email should appear in the Deleted/Trash folder

Tested in Unit Testing



Under Integration Testing

System Testing

What is System Testing?

- System testing is performed in the context of a System Requirement Specification (SRS) and/or a Functional Requirement Specifications (FRS). It is the final test to verify that the product to be delivered meets the specifications mentioned in the requirement document. It should investigate both functional and non-functional requirements.
- **System Testing** is a level of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system.
- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Testing of the user's experience with the application.

Verification Testing

- Verification Testing : is used to confirm that a product meets specifications or requirements as defined in Phase Zero of the product development process. Verification testing should be conducted iteratively throughout a product design process, ensuring that the designs perform as required by the product specifications.
- Product developers achieve verification using an array of methods that can include inspection, demonstration, physical testing, and simulation.
- Verification testing includes different activities such as business requirements, system requirements, design review, and code walkthrough while developing a product.
- It is also known as static testing, where we are ensuring that "**we are developing the right product or not**". And it also checks that the developed application fulfilling all the requirements given by the client.
- Verification testing – Did I make the product correctly?
- Validation testing – Did I make the correct product?

Validation Testing

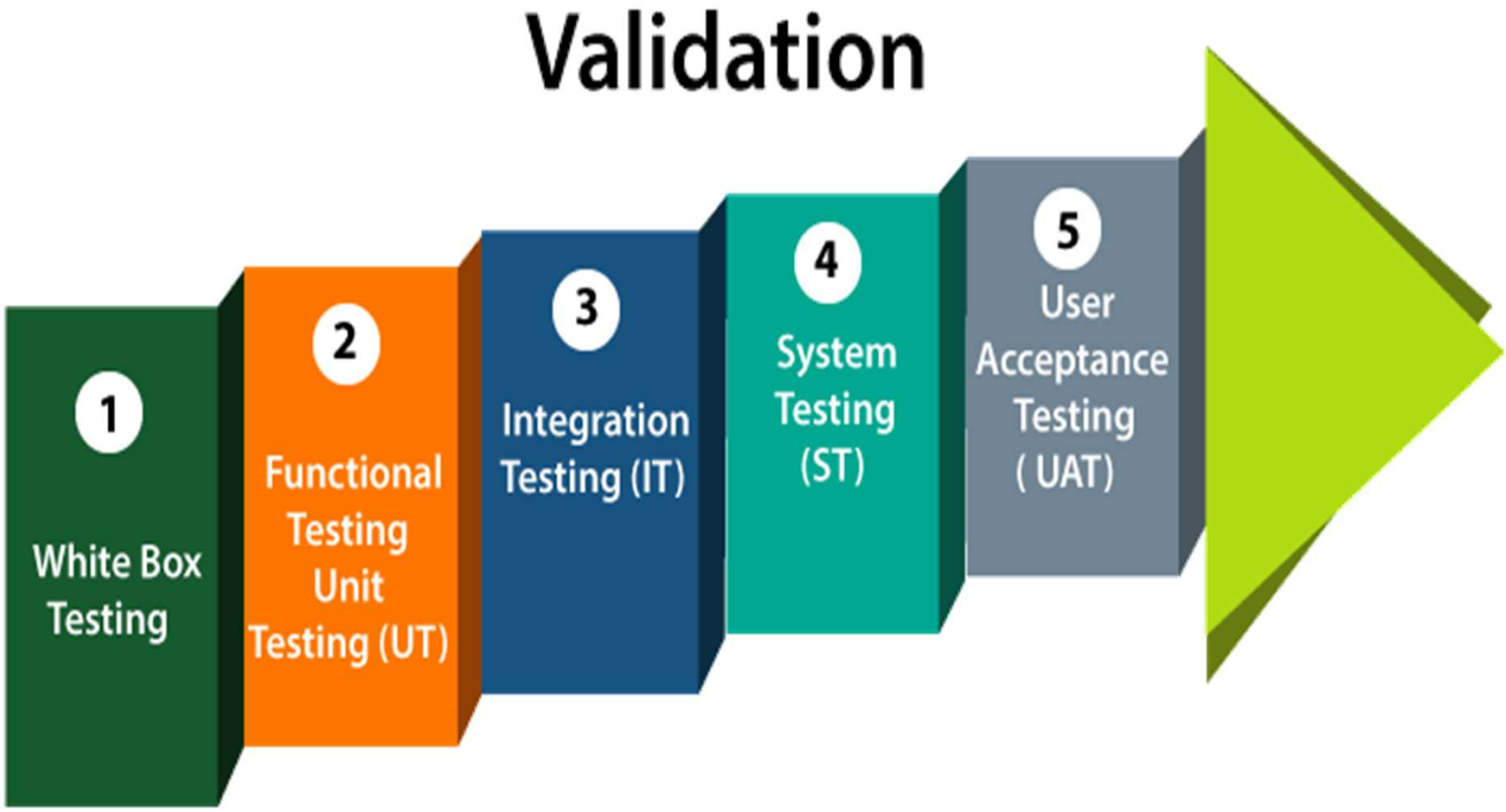
- Validation testing is the process of ensuring if the tested and developed software satisfies the client /user needs. The business requirement logic or scenarios have to be tested in detail. All the critical functionalities of an application must be tested here.

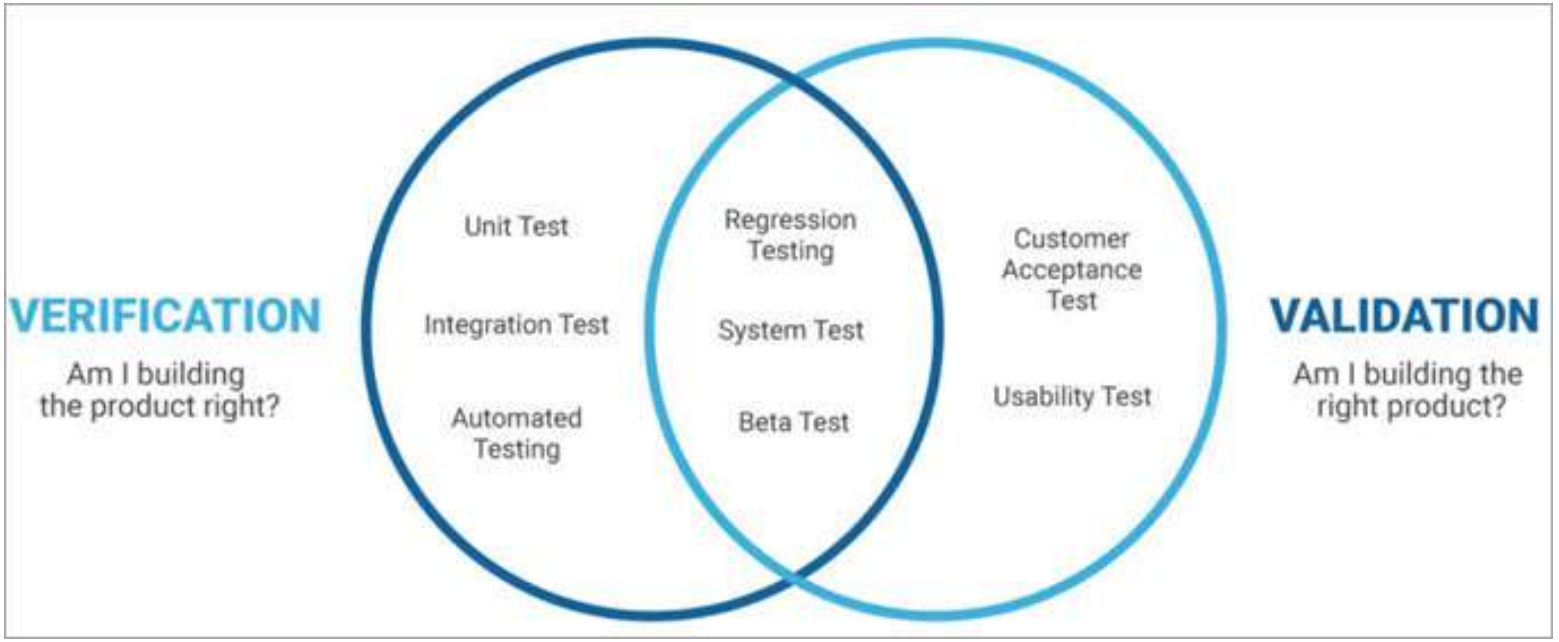
- Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.

- Verification and validation testing play an integral part in the product development process. Creating technology and systems is an iterative cycle, in which development teams are constantly learning and feeding data back into the process.

- Verification and validation activity serve as a system of checks and balances, ensuring that developers regularly evaluate if the product is functioning as intended and meeting the needs of its users. Testing also offers an important opportunity to document the tangible results of your design efforts.

Verification	Validation
We check whether we are developing the right product or not.	We check whether the developed product is right.
Verification is also known as **static testing**.	Validation is also known as **dynamic testing**.
Verification includes different methods like Inspections, Reviews, and Walkthroughs.	Validation includes testing like functional testing, system testing, integration and User acceptance testing.
It is a process of checking the work-products (not the final product) of a development cycle to decide whether the product meets the specified requirements.	It is a process of checking the software during or at the end of the development cycle to decide whether the software follow the specified business requirements.
Quality assurance comes under verification testing.	Quality control comes under validation testing.
The execution of code does not happen in the verification testing.	In validation testing, the execution of code happens.
In verification testing, we can find the bugs early in the development phase of the product.	In the validation testing, we can find those bugs, which are not caught in the verification process.
Verification testing is executed by the Quality assurance team to make sure that the product is developed according to customers' requirements.	Validation testing is executed by the testing team to test the application.
Verification is done before the validation testing.	After verification testing, validation testing takes place.
In this type of testing, we can verify that the inputs follow the outputs or not.	In this type of testing, we can validate that the user accepts the product or not.


Validation
1
White Box Testing
2
Functional Testing Unit Testing (UT)
3
Integration Testing (IT)
4
System Testing (ST)
5
User Acceptance Testing (UAT)


VERIFICATION
Am I building the product right?
Unit Test
Integration Test
Automated Testing
Regression Testing
System Test
Beta Test
Customer Acceptance Test
Usability Test
VALIDATION
Am I building the right product?

1. Usability Testing - To test if an application or product has good user experience or not.

2. Regression Testing - To confirm that a code change or addition has not adversely affected existing features.

3. Load Testing - It is a type of non-functional testing which helps understand the behaviour of the application under a specific expected load.

4. Functional Testing - It is a type of testing to verify that a product performs and functions correctly according to user specifications.

5. Migration Testing - Testing of programs used to migrate /convert data from one application to another replacement application.

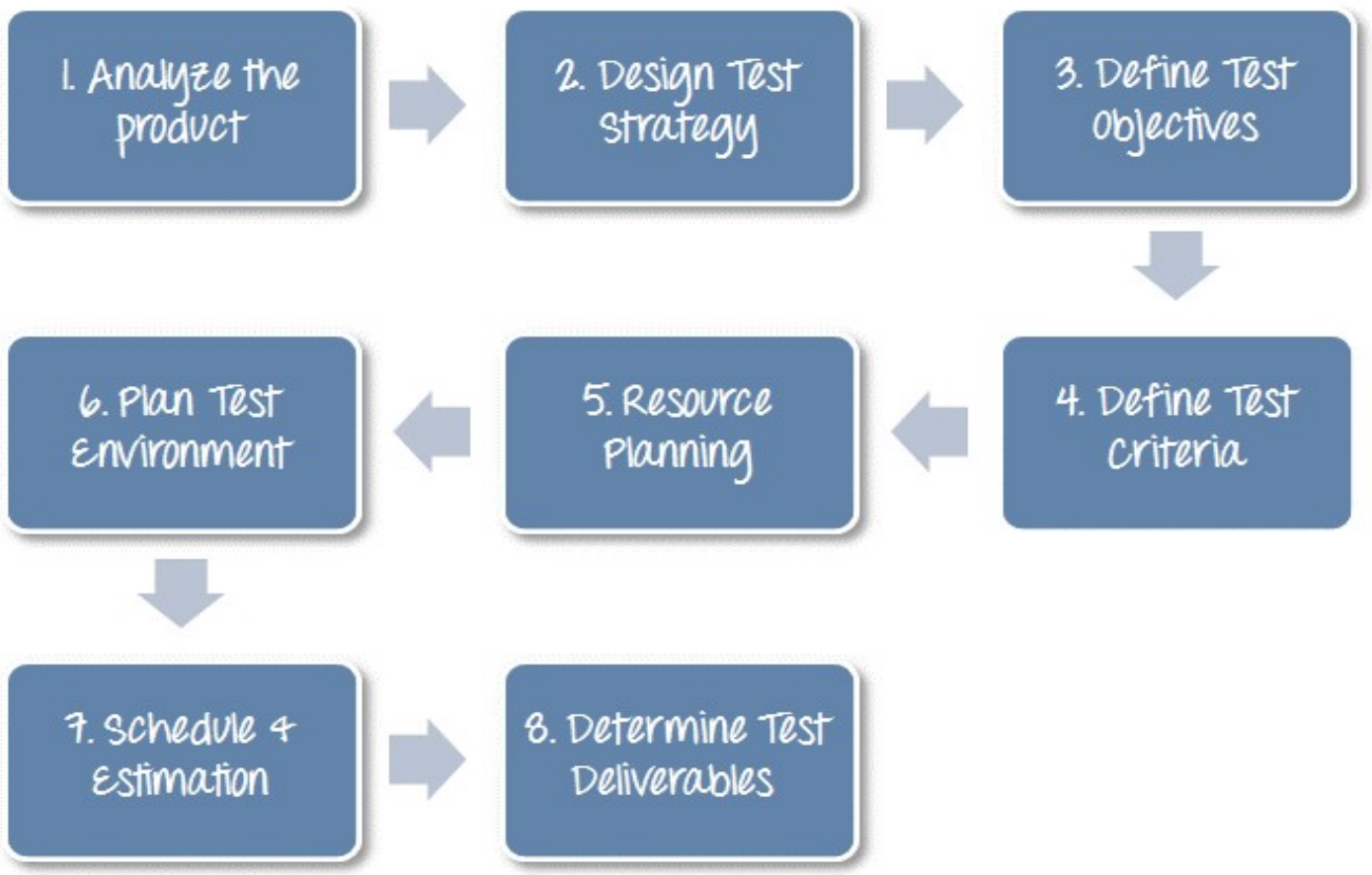
6. Compatibility Testing - It performed to validate that software performs same behaviour with different environment.

7. Boundary Value Testing - It is designed to include representatives of boundary values.

8. Fuzz Testing - It is used to provide invalid, unexpected, or random data to the inputs of a program.

Test plan

- A **Test Plan** is a detailed document that describes the test strategy, objectives, schedule, estimation, deliverables, and resources required to perform testing for a software product. Test Plan helps us determine the effort needed to validate the quality of the application under test. The test plan serves as a blueprint to conduct software testing activities as a defined process, which is minutely monitored and controlled by the test manager.
- **Test Plan is a dynamic document**. The success of a testing project depends upon a well-written Test Plan document that is current at all times. Test Plan is more or less like **a blueprint of how the testing activity is going** to take place in a project.
- **Given below are a few pointers on a Test Plan:**
  - Test Plan is a document that acts as a point of reference and only based on that testing is carried out within the QA team.
  - It is also a document that we share with the Business Analysts, Project Managers, Dev team and the other teams. This helps to enhance the level of transparency of the QA team's work to the external teams.
  - It is documented by the QA manager/QA lead based on the inputs from the QA team members.
  - Test Planning is typically allocated with 1/3rd of the time that takes for the entire QA engagement. The other 1/3rd is for Test Designing and the rest is for Test Execution.
  - This plan is not static and is updated on an on-demand basis.
  - The more detailed and comprehensive the plan is, the more successful will be the testing activity.


1. Analyze the product
2. Design Test Strategy
3. Define Test Objectives
4. Define Test Criteria
5. Resource Planning
6. Plan Test Environment
7. Schedule & Estimation
8. Determine Test Deliverables

1. Analyze the product
2. Design the Test Strategy
3. Define the Test Objectives
4. Define Test Criteria
5. Resource Planning
6. Plan Test Environment
7. Schedule & Estimation
8. Determine Test Deliverables

White Box Testing

- “White box testing” (also known as clear, glass box or structural testing) is a testing technique which evaluates the code and the internal structure of a program.
- White box testing involves looking at the structure of the code. When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification.
- White box testing in software engineering is based on the inner workings of an application and revolves around internal testing.
- By thoroughly examining the internal workings of the code, white box testing helps identify defects early in the development process, leading to higher-quality software products.

White Box Testing

Why we perform WBT?

- That all independent paths within a module have been exercised at least once.
- All logical decisions verified on their true and false values.
- All loops executed at their boundaries and within their operational bounds internal data structures validity.
- The term “Whitebox” was used because of the see-through box concept. The clear box or WhiteBox name symbolizes the ability to see through the software’s outer shell (or “box”) into its inner workings. Likewise, the “black box” in “Black Box Testing” symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

White Box Testing

Key aspects and techniques of white box testing include:

1. Code Coverage Analysis: This involves measuring the extent to which the source code has been executed during testing. It helps ensure that all parts of the code have been tested adequately. Common metrics for code coverage include statement coverage, branch coverage, path coverage, and condition coverage.

2. Control Flow Testing: This technique focuses on exercising different paths through the program's control structures, such as loops, conditionals, and branches. The goal is to test all possible control flow paths to uncover any potential errors in the logic of the code.

3. Data Flow Testing: This technique involves analyzing how data is input, manipulated, and output within the program. Test cases are designed to cover different data flows, including variable definitions, assignments, references, and usage, to identify potential data-related errors.

4. Branch Testing: This technique aims to test each possible branch or decision point within the code, ensuring that all possible outcomes are evaluated.

White Box Testing

5. Path Testing: Path testing involves testing all possible paths through the code, from the entry point to the exit point. This technique ensures that every possible sequence of statements is executed and evaluated.

6. Statement Testing: This technique focuses on testing individual statements within the code to ensure that each statement is functioning correctly.

7. Boundary Value Analysis: This technique involves testing the boundaries and edge cases of input ranges to uncover any potential errors or unexpected behavior at the extremes of valid input values.

8. Equivalence Partitioning: This technique involves dividing the input domain of a program into equivalence classes and then selecting representative values from each class as test cases. It helps reduce the number of test cases while still ensuring adequate coverage.

Control Structure

- Control structure testing is a group of white-box testing methods.
  - Branch Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing

Branch Testing

- also called Decision Testing
- definition: "For every decision, each branch needs to be executed at least once."
- shortcoming - ignores implicit paths that result from compound conditionals.
- Treats a compound conditional as a single statement. (We count each branch taken out of the decision, regardless which condition lead to the branch.)
- This example has two branches to be executed:

- **Data Flow Testing**
- Selects test paths according to the location of definitions and use of variables. This is a somewhat sophisticated technique and is not practical for extensive use. Its use should be targeted to modules with nested if and loop statements.
- **Loop Testing**
- Loops are fundamental to many algorithms and need thorough testing.
- There are four different classes of loops: simple, concatenated, nested, and unstructured.
- **Simple Loops,** where *n* is the maximum number of allowable passes through the loop.
  - Skip loop entirely
  - Only one pass through loop
  - Two passes through loop
  - m passes through loop where m<n.
  - (n-1), n, and (n+1) passes through the loop.
- **Nested Loops**
  - Start with inner loop. Set all other loops to minimum values.
  - Conduct simple loop testing on inner loop.
  - Work outwards
  - Continue until all loops tested.
- **Concatenated Loops**
  - If independent loops, use simple loop testing.
  - If dependent, treat as nested loops.
- **Unstructured loops**
  - Don't test - redesign.

Basis Path Testing

What is Path Testing?

- Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program.
- **Basis Path Testing** in software engineering is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program. The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.

Basis Path Testing

Steps for Basis Path testing

- The basic steps involved in basis path testing include
- Draw a control graph (to determine different program paths)
- Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- Find a basis set of paths
- Generate test cases to exercise each path

Basis Path Testing

Advantages :
Basis Path Testing can be applicable in the following cases:

1. **More Coverage –**
   Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.

2. **Maintenance Testing –**
   When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.

3. **Unit Testing –**
   When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.

4. **Integration Testing –**
   When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

5. **Testing Effort –**
   Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program

Black Box Testing

- **Blackbox testing, involves testing from an external or end-user type perspective.**
- Black box testing is a software testing technique that focuses on testing the functionality of a software application without knowledge of its internal code structure, design, or implementation details. In black box testing, the tester interacts with the software as an external user and evaluates its behavior based solely on its inputs and outputs, without considering its internal workings.
- Black box testing is advantageous because it does not require knowledge of the internal implementation details of the software, making it suitable for testing software developed by third-party vendors or when the internal code is complex or proprietary. It helps identify issues related to incorrect functionality, missing features, usability problems, and compatibility issues without needing to delve into the intricacies of the codebase.

Black Box Testing

Key aspects and techniques of black box testing include:

1. Functional Testing: Black box testing primarily focuses on verifying whether the software meets the specified functional requirements. Test cases are designed based on the expected behavior of the software as described in its requirements documentation.

2. Input-Output Testing: Test cases are designed to exercise various inputs to the software and evaluate the corresponding outputs or responses. This helps ensure that the software behaves as expected under different input conditions.

3. Boundary Value Analysis: This technique involves testing the software with input values at the boundaries of valid ranges, as well as just beyond those boundaries. It helps uncover potential errors or unexpected behavior related to boundary conditions.

4. Equivalence Partitioning: This technique involves dividing the input domain of the software into equivalence classes and selecting representative values from each class as test cases. It helps reduce the number of test cases while still ensuring adequate coverage.

Black Box Testing

5. Error Guessing: Testers use their intuition and experience to identify potential areas of weakness or vulnerability in the software and design test cases to specifically target those areas.

6. Ad Hoc Testing: Testers perform exploratory testing without following any predefined test cases or scripts. This allows them to uncover unexpected defects or behavior in the software.

7. Regression Testing: Black box testing is often used for regression testing, where previously tested functionality is retested after changes or enhancements to the software to ensure that existing functionality has not been negatively impacted.

8. Usability Testing: Testers evaluate the usability of the software from an end-user perspective, focusing on aspects such as ease of use, intuitiveness, and user interface design.

Black Box Testing

Steps of black box testing

- The black box test is based on the specification of requirements, so it is examined in the beginning.
- In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
- In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
- The fourth phase includes the execution of all test cases.
- In the fifth step, the tester compares the expected output against the actual output.
- In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Boundary Testing

- Boundary testing is a software testing technique used to evaluate the behaviour of a software application at its boundaries.

- The purpose of boundary testing is to ensure that the software handles input values at the extremes of valid ranges appropriately and to identify any errors or unexpected behaviour that may occur at these boundaries.

- Boundary testing is essential for ensuring that the software handles boundary conditions effectively and behaves as expected across the entire range of input values, thereby enhancing its overall quality and reliability.

Boundary Testing

Key aspects of boundary testing:

1. Identifying Boundaries: The first step in boundary testing is to identify the boundaries of input ranges, such as minimum and maximum values, limits, thresholds, and edge cases. These boundaries are often defined in the software requirements or specifications.

2. Boundary Conditions: Boundary conditions refer to the values immediately before and after the boundaries of input ranges. For example, if a range is defined as 1 to 10, the boundary conditions would include values such as 0, 1, 10, and 11.

3. Boundary Value Analysis (BVA): Boundary value analysis is a technique commonly used in boundary testing. It involves selecting test cases based on the boundary conditions and testing values at or just beyond these boundaries. The rationale behind BVA is that errors are more likely to occur at the boundaries of input ranges than within the ranges themselves.

4. Test Case Design: Test cases for boundary testing are designed to cover both sides of the boundary, including values just below and just above the boundary. For example, if the boundary is defined as 10, test cases might include inputs like 9, 10, and 11 to evaluate how the software behaves in each scenario.

Boundary Testing

5. Testing Single and Multiple Boundaries: Boundary testing may involve testing single boundaries (e.g., testing only the minimum or maximum value of a range) or multiple boundaries (e.g., testing both the minimum and maximum values). Testing multiple boundaries helps ensure comprehensive coverage of the input range.

6. Expected Results: Testers analyze the expected behavior of the software at each boundary and compare it against the actual behavior observed during testing. Any discrepancies or deviations from the expected results are noted as potential defects.

7. Error Handling: Boundary testing helps uncover potential errors related to boundary conditions, such as off-by-one errors, boundary overflow, boundary underflow, and boundary-related constraints violations. The software should handle these errors gracefully and provide appropriate error messages or responses to users.

8. Validation and Verification: Boundary testing is part of the validation process to verify that the software meets the specified requirements and behaves correctly under boundary conditions. It helps improve the robustness, reliability, and quality of the software by identifying and addressing boundary-related issues early in the development lifecycle.

Sandwich Testing

Sandwich testing, also known as sandwich integration testing or sandwich testing strategy, is a software testing approach that combines elements of both top-down and bottom-up integration testing methodologies. In sandwich testing, modules or components are tested in layers, with some integration testing occurring from the top down, and some from the bottom up, meeting in the middle, hence the term "sandwich."

Key Aspects sandwich testing :

1. Top-Down Integration Testing: In the top-down approach, testing begins with the higher-level modules or components, which are tested first. These higher-level modules are often the ones that have more dependencies on other modules and are critical to the overall functionality of the system. Stub modules or drivers may be used to simulate the behavior of lower-level modules that have not yet been implemented. Top-down integration testing continues until the lower-level modules are reached.

2. Bottom-Up Integration Testing: In contrast, bottom-up integration testing starts with the lower-level modules, which are tested independently or in small groups. These lower-level modules are usually less dependent on other modules and are tested without their dependencies. As the testing progresses, higher-level modules are gradually added and tested until the entire system is integrated.

Sandwich Testing

3. Meeting in the Middle: The sandwich testing strategy involves both top-down and bottom-up integration testing simultaneously. As testing progresses, the integration of modules from both ends of the system meets in the middle. This approach helps identify integration issues and defects earlier in the testing process and provides a more balanced and comprehensive coverage of the system's integration points.

4. Integration Points: Throughout the sandwich testing process, integration points between modules are thoroughly tested to ensure that data and control flow seamlessly between different parts of the system. This includes testing data exchange, function calls, error handling, and communication protocols between modules.

5. Incremental Testing: Sandwich testing is often conducted incrementally, with testing and integration activities performed iteratively as new modules are developed and integrated into the system. This incremental approach allows for early detection and resolution of integration issues, reducing the risk of major defects in the final product.

6. Regression Testing: As modules are integrated and tested incrementally, regression testing is performed to ensure that changes or additions to the system do not introduce new defects or disrupt existing functionality. Regression testing helps maintain the integrity and stability of the system throughout the development process.

Overall, sandwich testing offers a balanced integration testing approach by combining the strengths of both top-down and bottom-up methodologies. By identifying and addressing integration issues early, sandwich testing helps improve the quality, reliability, and maintainability of software systems.

Inventory Analysis

In software engineering, inventory analysis refers to the process of assessing and managing the various assets and components used in software development and maintenance. This includes analyzing the inventory of software artifacts, such as source code, libraries, dependencies, documentation, and other resources, to ensure efficient management, optimization, and control of software-related assets.

Following key components of inventory analysis in software engineering:

1. Source Code Inventory: Inventory analysis involves cataloging and organizing the source code files and modules used in software development projects. This includes identifying the programming languages, frameworks, and technologies utilized, as well as documenting dependencies and interdependencies between different code components.

2. Library and Dependency Management: Software projects often rely on third-party libraries, frameworks, and dependencies to implement certain functionalities or features. Inventory analysis includes tracking and managing these external dependencies, ensuring that they are up-to-date, compatible with other components, and properly licensed.

Inventory Analysis

3. Documentation Inventory: Documentation is an essential part of software development, providing guidance, instructions, and information about the software architecture, design, functionality, and usage. Inventory analysis involves maintaining an inventory of documentation assets, such as design documents, user manuals, API references, release notes, and technical specifications, to support the development, maintenance, and usage of the software.

4. Version Control and Configuration Management: Inventory analysis encompasses managing the versioning and configuration of software artifacts using version control systems (e.g., Git, Subversion). This includes tracking changes, revisions, and branches in the source code, as well as managing configurations for different environments (e.g., development, testing, production).

5. Asset Tracking and Traceability: Inventory analysis involves tracking and tracing software assets throughout their lifecycle, from initial development and testing to deployment and maintenance. This includes maintaining a record of changes, updates, and releases for each software component, as well as documenting dependencies and relationships between different assets.

Inventory Analysis

6. License Compliance and Governance: Inventory analysis includes ensuring compliance with software licenses and intellectual property rights for third-party components and libraries used in software projects. This involves tracking license information, restrictions, and obligations, as well as implementing governance policies and procedures to mitigate legal and regulatory risks.

7. Optimization and Resource Management: Based on the findings of inventory analysis, software engineering teams can identify opportunities for optimization and resource management. This may include reducing code duplication, eliminating unused or obsolete components, streamlining dependencies, and improving overall software quality and performance.

8. Risk Management and Security: Inventory analysis includes assessing and managing risks related to software assets, such as security vulnerabilities, code quality issues, and potential compliance violations. This involves implementing security measures, code reviews, and testing practices to mitigate risks and ensure the integrity and security of software assets.

Overall, inventory analysis is essential for effective software asset management, enabling organizations to optimize resource utilization, mitigate risks, ensure compliance, and improve overall software development and maintenance processes. It enables the software engineering teams to better manage complexity, enhance collaboration, and deliver high-quality software products and services

Maintenance Log

Creating a maintenance log in software engineering involves several steps to ensure that all maintenance activities, including bug fixes, enhancements, updates, and other modifications to the software, are properly documented and tracked over time. Here are the different steps involved in creating a maintenance log:

1. Define Maintenance Categories: Determine the categories or types of maintenance activities that will be logged. Common categories include bug fixes, feature enhancements, performance optimizations, security updates, and technical debt reduction.

2. Choose a Format: Decide on the format and structure of the maintenance log. This could be a spreadsheet, a database, a document, or a dedicated software tool. Consider factors such as ease of use, accessibility, and scalability when choosing the format.

3. Define Log Fields: Identify the essential information that needs to be captured for each maintenance entry. This typically includes details such as the date of the maintenance activity, a brief description of the change, the affected component or feature, the reason for the change, the assigned developer or team, the status of the maintenance task, and any relevant comments or notes.

Maintenance Log

4. Capture Maintenance Entries: Record maintenance activities as they occur, ensuring that each entry in the maintenance log is accurate and complete. Include details such as the date and time of the maintenance activity, the nature of the change, any relevant references (e.g., bug IDs, feature requests), and the individuals involved in the maintenance task.

5. Track Changes Over Time: Maintain a chronological record of maintenance activities, allowing stakeholders to track changes to the software over time. Use timestamps to indicate when each maintenance entry was created or updated, enabling historical analysis and trend identification.

6. Assign Responsibilities: Assign responsibilities for updating and maintaining the maintenance log to designated individuals or teams within the software development organization. Ensure that all relevant stakeholders are aware of their roles and responsibilities regarding maintenance log management.

7. Regularly Update the Log: Keep the maintenance log up-to-date by regularly updating it with new maintenance entries and status updates. Ensure that all maintenance activities, regardless of their size or significance, are documented in the log to maintain a comprehensive record of changes to the software.

Maintenance Log

9. Accessibility and Transparency: Make the maintenance log accessible to relevant stakeholders, such as developers, testers, project managers, and customers, as appropriate. Transparency regarding maintenance activities fosters collaboration, accountability, and trust within the software development team and with external stakeholders.

10. Use Reporting and Analysis: Utilize reporting and analysis tools to generate insights from the maintenance log data. Identify patterns, trends, and areas for improvement in the software maintenance process based on the information captured in the log. Use this analysis to inform decision-making and drive continuous improvement efforts.

Performance Testing

Performance testing in software engineering is a type of testing that evaluates how a software application performs under various conditions, such as workload, concurrency, and stress. The primary objective of performance testing is to assess the speed, responsiveness, scalability, reliability, and resource usage of the software under different scenarios, ensuring that it meets the performance requirements and user expectations.

Key aspects and objectives of performance testing:

1. Types of Performance Testing:

- Load Testing: Load testing involves testing the software's performance under expected load conditions, such as a normal number of users or transactions. It helps identify performance bottlenecks and assesses whether the software can handle the expected workload.

- Stress Testing: Stress testing evaluates the software's behavior under extreme conditions, such as high traffic volumes, excessive data loads, or resource constraints. It helps determine the software's breaking point and assess its robustness and resilience under stressful conditions.

- Endurance Testing: Endurance testing, also known as soak testing, involves running the software under a sustained load for an extended period to evaluate its stability and performance over time. It helps identify memory leaks, resource exhaustion, and other long-term performance issues.

Performance Testing

Types of Performance Testing continued:

- Scalability Testing: Scalability testing assesses how well the software can scale to accommodate increased load or user demand. It helps determine whether the software can handle growing volumes of data, users, or transactions without degradation in performance.

- Volume Testing: Volume testing evaluates the software's performance with large volumes of data, such as database records, files, or transactions. It helps identify performance issues related to data processing, storage, and retrieval.

- Concurrency Testing: Concurrency testing assesses how well the software handles simultaneous user interactions or concurrent transactions. It helps identify synchronization issues, race conditions, and contention for shared resources.

2. Performance Metrics:

- Response Time: The time taken for the software to respond to a user request or transaction.
- Throughput: The rate at which the software can process transactions or data.
- Concurrency: The number of users or transactions the software can handle simultaneously.
- Resource Utilization: The usage of system resources such as CPU, memory, disk I/O, and network bandwidth.
- Error Rate: The frequency of errors or failures encountered during testing.
- Scalability Limits: The maximum load or capacity the software can handle before performance degrades.

Performance Testing

3. Test Environment Setup:

- Establishing a test environment that closely resembles the production environment in terms of hardware, software, network configuration, and user behavior.
- Configuring monitoring tools and performance profiling instruments to capture performance metrics during testing.

4. Test Scenario Design:

- Defining test scenarios and workload profiles based on expected user behavior, usage patterns, and system requirements.
- Identifying critical use cases, peak load scenarios, and stress conditions to simulate during testing.

5. Test Execution:

- Executing performance tests according to predefined test scenarios and workload profiles.
- Monitoring and measuring performance metrics in real-time during test execution.
- Analyzing test results to identify performance bottlenecks, resource constraints, and areas for optimization.

Performance Testing

6. Performance Tuning and Optimization:

- Identifying and addressing performance issues through code optimization, database tuning, caching strategies, and infrastructure scaling.
- Iteratively retesting and validating performance improvements to ensure that they meet the desired performance objectives.

7. Reporting and Analysis:

- Documenting performance test results, including metrics, observations, and recommendations.
- Communicating findings to stakeholders, including developers, testers, project managers, and business owners.
- Iteratively refining performance testing strategies based on feedback and lessons learned from previous testing cycles.

- Overall, performance testing plays a crucial role in ensuring that software applications meet performance expectations, deliver optimal user experience, and perform reliably under various operating conditions.
- By proactively identifying and addressing performance issues early in the development lifecycle, organizations can minimize the risk of performance-related problems in production and enhance the overall quality and performance of their software products.

Polymorphism Testing

- **Polymorphism testing** in software engineering refers to the process of testing software components, particularly object-oriented programs, to ensure that they correctly exhibit polymorphic behavior.

- Polymorphism is a fundamental concept in object-oriented programming (OOP), where objects of different classes can be treated uniformly through a common interface. Polymorphism testing verifies that objects behave as expected when invoked through interfaces or base classes, even when they are instances of derived classes.

Key aspects and objectives of polymorphism testing:

1. Understanding Polymorphism: Before diving into polymorphism testing, it's essential to understand polymorphism itself. In OOP, polymorphism allows objects of different classes to be treated as instances of a common superclass or interface. This enables code reuse, flexibility, and extensibility in software design.

2. Identifying Polymorphic Behavior: Polymorphism testing focuses on testing scenarios where polymorphic behavior is expected or required. This includes scenarios where methods or operations are invoked through base class references or interface implementations, allowing for dynamic binding and runtime method resolution.

3. Test Scenario Design: Design test scenarios that exercise polymorphic behavior by invoking methods or operations through base class references or interface implementations. These scenarios should cover various combinations of object types and method calls to ensure comprehensive coverage of polymorphic interactions.

Polymorphism Testing

4. Test Case Development: Develop test cases that validate the expected behavior of polymorphic objects under different conditions. Test cases should include inputs that trigger different code paths and behaviors in derived classes, as well as edge cases and boundary conditions.

5. Test Execution: Execute the test cases against the software components to verify that polymorphic behavior is correctly exhibited. This involves invoking methods or operations through base class references or interface implementations and comparing the actual behavior against the expected behavior defined in the test cases.

6. Handling Inheritance and Overrides: Pay special attention to scenarios involving inheritance and method overrides, as polymorphism behavior can be influenced by subclass implementations overriding methods defined in superclass or interface declarations. Test cases should validate that overridden methods are invoked correctly and produce the expected results.

7. Dynamic Binding and Runtime Dispatch: Verify that dynamic binding and runtime dispatch mechanisms correctly resolve method calls at runtime based on the actual types of objects involved. This ensures that the appropriate method implementations are invoked dynamically, based on the runtime type of the object.

Polymorphism Testing

8. Edge Cases and Error Handling: Test edge cases and error conditions to ensure that polymorphic behavior is handled correctly under exceptional circumstances. This includes scenarios such as null references, invalid inputs, and unexpected runtime conditions.

9. Integration Testing: Integrate polymorphism testing into the broader testing strategy, including integration testing, system testing, and regression testing. Polymorphism issues can manifest at various levels of system integration, so it's crucial to validate polymorphic behavior in the context of larger system interactions.

10. Documentation and Reporting: Document test results, including observed behaviors, deviations from expected behavior, and any defects or anomalies encountered during testing. Report findings to stakeholders and collaborate with developers to address and resolve any identified polymorphism issues.

- Overall, polymorphism testing plays a critical role in ensuring the correctness, robustness, and maintainability of object-oriented software systems by validating the expected behavior of polymorphic objects and interactions.
- By systematically testing polymorphic behavior, software engineers can build confidence in the reliability and consistency of their software components and ensure that they function correctly across different object types and class hierarchies.

Code Restructuring

Code restructuring, or code refactoring, is the process of improving existing code without changing its external behavior.

It involves making changes to enhance readability, maintainability, and performance, while reducing complexity and technical debt.

This process helps ensure that the codebase remains easy to understand, modify, and extend as the software evolves.

Some common reasons for code restructuring

1.Improving Readability.

2.Enhancing Maintainability

3.Optimizing Performance

4.Ensuring Consistency

5.Supporting Evolution

6.Reducing Technical Debt

Forward Engineering

Forward engineering in software engineering refers to the traditional process of starting with requirements and designing and implementing a system to fulfill those requirements.

It's essentially the typical software development lifecycle where you begin with analyzing requirements, proceed to design, then implementation, testing, and finally deployment.

Various steps in forward engineering process:

1.Requirements Analysis: This is the initial phase where the stakeholders' needs and requirements are gathered and documented. It involves understanding the problem domain, identifying functional and non-functional requirements, and establishing goals for the software system.

2.System Design: In this phase, based on the requirements gathered, the system architecture is designed. This includes defining the overall structure of the system, its components/modules, their relationships, and the interfaces between them. Different design methodologies like object-oriented design or component-based design may be employed.

3.Implementation: Once the design is finalized, the actual coding of the software begins. Developers write the source code according to the design specifications. This phase involves translating the design into a programming language and constructing the software system.

Forward Engineering

4. **Testing**: After the implementation phase, rigorous testing is performed to identify and rectify any defects or bugs in the software. Various testing techniques such as unit testing, integration testing, system testing, and acceptance testing are employed to ensure the quality and reliability of the software.
5. **Deployment**: Once the software has been thoroughly tested and deemed ready for release, it is deployed to the production environment. Deployment involves installing the software on the target hardware and making it available for use by end-users.

Forward engineering is often contrasted with reverse engineering, where the process involves analyzing an existing system to understand its functionality, structure, and behavior, typically for purposes such as maintenance, enhancement, or reengineering.

Reverse Engineering

- Reverse engineering is the process of analyzing a product, system, or component to understand its structure, functionality, and behavior, often without access to its original design documentation or source code.
- This practice is commonly employed in various fields, including software engineering, mechanical engineering, electronics, and more.

Here's an analysis of reverse engineering:

1.Purpose: Reverse engineering is typically performed for several reasons:

- Understanding: To gain insights into how a product or system works.
- Interoperability: To ensure compatibility with other systems or components.
- Improvement: To enhance or modify existing products or systems.
- Compatibility: To develop replacements or alternative solutions for legacy or proprietary systems.
- Security: To identify vulnerabilities or weaknesses in software or hardware systems.

2. Process:

- Information Gathering: Collecting data about the system, which may include examining the system's behavior, dissecting its components, analyzing its inputs and outputs, and studying any available documentation.
- Decompilation/Disassembly: In software engineering, this involves converting machine code or executable binaries back into a higher-level programming language or assembly code to understand its logic and algorithms.
- Analysis: Examining the extracted information to understand the system's structure, relationships, and functionalities. This may involve identifying patterns, algorithms, data structures, and communication protocols.
- Documentation: Documenting the findings and insights gained during the reverse engineering process to create a comprehensive understanding of the system for future reference or modification.

Reverse Engineering

3. **Tools**:
   - Reverse engineering tools vary depending on the type of system being analyzed. For software, tools like disassemblers, decompilers, debuggers, and code analysis utilities are commonly used.
   - Hardware reverse engineering may involve tools such as oscilloscopes, logic analyzers, hardware debuggers, and reverse engineering software tools.

4.Legal and Ethical Considerations:
   - Reverse engineering can raise legal and ethical concerns, especially when it involves proprietary or copyrighted systems.
   - Intellectual property laws and end-user license agreements may restrict or prohibit reverse engineering in certain contexts.
   - Ethical considerations include respecting the rights of original creators, avoiding unauthorized access to systems, and adhering to professional standards and guidelines.

5. **Challenges**:
   - Reverse engineering can be time-consuming and resource-intensive, particularly for complex systems.
   - It may be challenging to accurately reconstruct the original design or functionality, especially without access to comprehensive documentation.
   - Legal and ethical considerations, as mentioned earlier, can pose significant challenges and constraints.

Reverse engineering is a valuable process for understanding, analyzing, and modifying existing systems or components, but it requires careful consideration of legal, ethical, and technical factors.

Forward engineering	Reverse engineering
Applications are developed with given requirements	Information for the system are collected from the given application
It's flow is model => System	It's flow is System => model
Takes more time for development	Takes less time for development
Prescriptive, Developers are told how to work.	Adaptive, Engineer must find out what actually the developer did
Production is started with given requirements.	Production is started by taking existing product.
It requires high proficiency skills	It may not require high proficiency skills
For example:- Developing a new software from scratch	For example:- Cloning Facebook, Instagram, Paypal

Inspection

- Inspections in software engineering refer to a formalized process of reviewing software artifacts, such as requirements documents, designs, or code, to identify defects, errors, or inconsistencies early in the development lifecycle.
- Inspections are a proactive quality assurance technique aimed at improving the quality of software products.
- Explanation of inspections steps along with an example:

1. Planning:

- **Objective Definition**: Define the objectives of the inspection, such as finding defects, ensuring compliance with standards, or verifying adherence to requirements.
- **Artifact Selection**: Select the software artifact(s) to be inspected, such as requirements documents, design specifications, or source code.
- **Composition of Inspection Team**: Assemble a team of reviewers with diverse expertise relevant to the artifact being inspected.

2. Preparation:

- **Individual Preparation**: Each reviewer independently examines the artifact to be inspected, noting potential defects or areas of concern.
- **Checklist Creation**: Develop a checklist or guideline to ensure consistency and thoroughness in the inspection process. The checklist may include common types of defects, adherence to coding standards, or compliance with requirements.

Inspection

3. Inspection Meeting:

- **Moderator**: A moderator leads the inspection meeting, guiding the review process and ensuring that it stays focused and productive.
- **Presentation**: The author of the artifact presents it to the inspection team, providing context, explanations, and clarifications as needed.
- **Defect Identification**: Reviewers discuss the artifact, identify defects, inconsistencies, or areas requiring improvement, and document them.

4. Rework:

- **Defect Correction**: The author addresses the identified defects and incorporates necessary changes into the artifact.
- **Verification**: Reviewers verify that the corrections have been made satisfactorily and that the artifact now meets the required quality standards.

5. Follow-Up:

- **Documentation**: Record the outcomes of the inspection, including identified defects, corrective actions taken, and lessons learned.
- **Process Improvement**: Use insights gained from the inspection to improve future development processes, such as refining coding standards, enhancing requirements elicitation techniques, or providing additional training for team members.

Inspection

Example:

Consider a software development team working on an e-commerce website. Before starting the implementation phase, the team decides to conduct an inspection of the system's design documents. They select the architectural design document as the artifact to be inspected.

- **Planning**: The objective of the inspection is to ensure that the architectural design adequately addresses the system's functional and non-functional requirements while adhering to architectural principles and best practices. The inspection team comprises software architects, developers, and quality assurance engineers.
- **Preparation**: Each member of the inspection team independently reviews the architectural design document, looking for potential design flaws, inconsistencies, or violations of architectural principles. They use a checklist that includes items such as adherence to design patterns, scalability considerations, and modularity.
- **Inspection Meeting**: The team holds a meeting moderated by a software architect. The author of the architectural design document presents it to the team, explaining the rationale behind design decisions and addressing any questions or concerns raised by the reviewers. During the discussion, the team identifies several design flaws, such as tight coupling between modules and inadequate error handling mechanisms.

Inspection

- **Rework**: The author of the architectural design document revises it based on the identified defects, making necessary changes to improve the design's quality and clarity. The revised document is then re-inspected to ensure that all identified issues have been addressed satisfactorily.
- **Follow-Up**: The outcomes of the inspection, including identified design flaws and corrective actions taken, are documented for future reference. The team also discusses ways to prevent similar issues in future design documents, such as conducting design reviews at earlier stages of the development process or providing additional training on architectural design principles.

In summary, inspections are a systematic and collaborative approach to identifying and correcting defects in software artifacts, helping to improve overall product quality and reduce the likelihood of costly errors during later stages of development.

Objectives of Testing

Objectives for Testing:

1.Verification of Requirements: Testing ensures that the software system meets the specified requirements. By validating against user needs and expectations, it helps in building a product that satisfies customer demands.

2.Validation of Functionality: Testing verifies that all the functions and features of the software perform as intended. It ensures that the software behaves correctly under different scenarios and inputs.

3.Identification of Defects: One of the primary objectives of testing is to identify defects or bugs in the software. Detecting and fixing these issues early in the development lifecycle helps in delivering a more reliable and robust product.

4.Assessment of Quality: Testing provides insights into the quality of the software. It evaluates factors such as reliability, usability, performance, and security to ensure that the software meets the desired quality standards.

5.Risk Mitigation: Testing helps in identifying and mitigating risks associated with software development. By uncovering potential issues early, it reduces the likelihood of encountering major problems during production or deployment.

6.Improvement of Maintenance: Testing contributes to the long-term maintainability of the software by ensuring that it is well-structured, easy to understand, and simple to modify or extend.

Maintenance and Its Types

Maintenance in software engineering refers to the process of modifying, updating, and enhancing a software product after its initial development and deployment. It encompasses various activities aimed at ensuring that the software continues to meet the changing needs of users and remains functional, reliable, and efficient over its entire lifecycle. Maintenance is a critical phase in the software development process, accounting for a significant portion of the total cost and effort invested in a software project. Here's an explanation of maintenance along with its types:

Maintenance Activities:

1.Corrective Maintenance: This type of maintenance involves addressing defects, errors, or bugs identified in the software after it has been deployed. Corrective maintenance aims to diagnose and fix issues to restore the software to its desired functionality. It typically involves troubleshooting, debugging, and patching software to resolve issues reported by users or discovered during testing.

2.Adaptive Maintenance: Adaptive maintenance involves modifying the software to accommodate changes in the external environment, such as changes in hardware, operating systems, or regulatory requirements. This type of maintenance ensures that the software remains compatible with evolving technologies and remains operational in the face of environmental changes.

3.Perfective Maintenance: Perfective maintenance focuses on improving or enhancing the software to meet the evolving needs and preferences of users. It involves adding new features, optimizing performance, enhancing usability, and improving overall quality to enhance the software's functionality and user experience.

4.Preventive Maintenance: Preventive maintenance aims to proactively identify and address potential issues or weaknesses in the software before they manifest as problems. It involves activities such as code refactoring, performance tuning, and security updates to prevent future failures, improve reliability, and reduce the likelihood of system downtime or data loss.

Maintenance and Its Types

Types of Maintenance:

1.Corrective Maintenance: As mentioned earlier, corrective maintenance involves fixing defects or errors discovered in the software after deployment. This type of maintenance is reactive in nature, addressing issues as they arise to ensure the continued operation of the software.

2.Adaptive Maintenance: Adaptive maintenance involves modifying the software to adapt to changes in its operating environment, such as hardware upgrades, changes in operating systems, or compliance with new regulations. This type of maintenance is necessary to ensure that the software remains compatible and functional in changing circumstances.

3.Perfective Maintenance: Perfective maintenance focuses on enhancing the software's functionality, performance, and usability to meet the evolving needs of users. This type of maintenance involves adding new features, improving existing functionality, and optimizing performance to enhance the overall quality and user experience of the software.

4.Preventive Maintenance: Preventive maintenance aims to identify and address potential issues or weaknesses in the software before they result in failures or problems. This type of maintenance involves proactive measures such as code refactoring, performance tuning, and security updates to prevent future issues and ensure the long-term reliability and stability of the software.

By effectively managing and performing these types of maintenance activities, software organizations can ensure that their products remain relevant, reliable, and valuable to users throughout their lifecycle.

Scalability

Scalability refers to the ability of a system to handle an increasing amount of workload or to accommodate growth without sacrificing performance, reliability, or user experience. In the context of software, scalability typically refers to two aspects:

1.Horizontal Scalability: This involves adding more resources or instances of the software to distribute the workload across multiple machines or servers. For example, a web application might be designed to scale horizontally by adding more web servers to handle increasing traffic.

2.Vertical Scalability: This involves increasing the capacity of individual resources, such as upgrading the hardware or software components of a single server to handle greater load. For example, upgrading the CPU, memory, or storage capacity of a database server to support more concurrent users.

Scalability is crucial for ensuring that software systems can accommodate growth in terms of user base, data volume, or transaction volume without experiencing performance degradation or system failures.

Regression

- Regression in software refers to the unintended introduction of new defects or bugs in a software application as a result of changes made to the codebase. Regression testing is the process of retesting the software to ensure that new changes have not adversely affected the existing functionality.
- When developers modify or add new features to a software application, there is a risk that these changes may unintentionally introduce bugs or disrupt existing functionality. Regression testing aims to identify and detect such regressions by re-executing previously developed and executed test cases.
- The term "regression" comes from the idea that the software should "regress" to its previous state of functionality after changes are made. Regression testing helps ensure that the software remains stable and reliable, even as it undergoes continuous development and updates.


THANK YOU!