Software Testing Strategies

Software testing is the process of evaluating a software application to identify if it meets specified requirements and to identify any defects. **The following are common testing strategies:**

1. Black box testing- Tests the functionality of the software without looking at the internal code structure.

1. White box testing - Tests the internal code structure and logic of the software.

1. Unit testing - Tests individual units or components of the software to ensure they are functioning as intended.

1. Integration testing- Tests the integration of different components of the software to ensure they work together as a system.

1. Functional testing- Tests the functional requirements of the software to ensure they are met.

1. System testing- Tests the complete software system to ensure it meets the specified requirements.

1. Acceptance testing - Tests the software to ensure it meets the customer's or end-user's expectations.

1. Regression testing - Tests the software after changes or modifications have been made to ensure the changes have not introduced new defects.

1. Performance testing - Tests the software to determine its performance characteristics such as speed, scalability, and stability.

1. Security testing- Tests the software to identify vulnerabilities and ensure it meets security requirements.

Software Testing is a type of investigation to find out if there are any defects or errors present in the software so that the errors can be reduced or removed to increase the quality of the software and to check whether it fulfills the specified requirements or not.
According to Glen Myers, software testing has the following objectives:

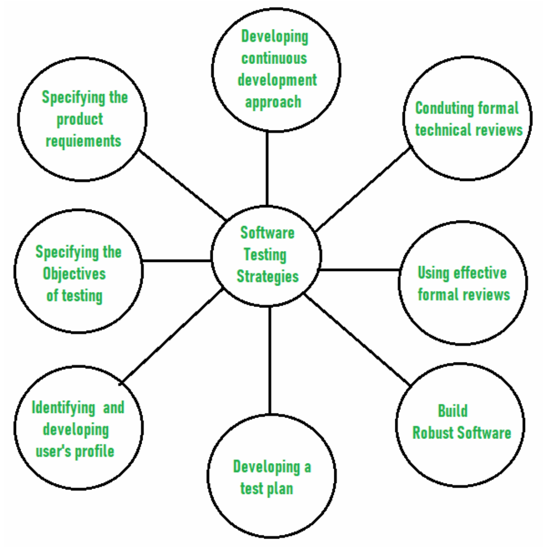
- The process of investigating and checking a program to find whether there is an error or not and does it fulfills the requirements or not is called testing.

- When the number of errors found during the testing is high, it indicates that the testing was good and is a sign of a good test case.

- Finding an unknown error that wasn't discovered yet is a sign of a successful and good test case.

Software testing Strategies

The main **objective** of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software. The overall strategy for testing software includes:


Developing continuous development approach
Specifying the product requiements
Conduting formal technical reviews
Software Testing Strategies
Specifying the Objectives of testing
Using effective formal reviews
Identifying and developing user's profile
Build Robust Software
Developing a test plan

Software testing Strategies

Before testing starts, it's necessary to identify and specify the requirements of the product in a quantifiable manner. Different characteristics quality of the software is there such as maintainability that means the ability to update and modify, the probability that means to find and estimate any risk, and usability that means how it can easily be used by the customers or end-users. All these characteristic qualities should be specified in a particular order to obtain clear test results without any error.

1. **Specifying the objectives of testing in a clear and detailed manner.** Several objectives of testing are there such as effectiveness that means how effectively the software can achieve the target, any failure that means inability to fulfill the requirements and perform functions, and the cost of defects or errors that mean the cost required to fix the error. All these objectives should be clearly mentioned in the test plan.

1. **For the software, identifying the user's category and developing a profile for each user.** Use cases describe the interactions and communication among different classes of users and the system to achieve the target. So as to identify the actual requirement of the users and then testing the actual use of the product.

1. **Developing a test plan to give value and focus on rapid-cycle testing.** Rapid Cycle Testing is a type of test that improves quality by identifying and measuring the any changes that need to be required for improving the process of software. Therefore, a test plan is an important and effective document that helps the tester to perform rapid cycle testing.

1. **Robust software is developed that is designed to test itself.** The software should be capable of detecting or identifying different classes of errors. Moreover, software design should allow automated and regression testing which tests the software to find out if there is any adverse or side effect on the features of software due to any change in code or program.

1. **Before testing, using effective formal reviews as a filter.** Formal technical reviews is technique to identify the errors that are not discovered yet. The effective technical reviews conducted before testing reduces a significant amount of testing efforts and time duration required for testing software so that the overall development time of software is reduced.

1. **Conduct formal technical reviews to evaluate the nature, quality or ability of the test strategy and test cases.** The formal technical review helps in detecting any unfilled gap in the testing approach. Hence, it is necessary to evaluate the ability and quality of the test strategy and test cases by technical reviewers to improve the quality of software.

1. **For the testing process, developing a approach for the continuous development.** As a part of a statistical process control approach, a test strategy that is already measured should be used for software testing to measure and control the quality during the development of software.

Advantages of Software Testing

- **Improves software quality and reliability** - Testing helps to identify and fix defects early in the development process, reducing the risk of failure or unexpected behavior in the final product.
- **Enhances user experience** - Testing helps to identify usability issues and improve the overall user experience.
- **Increases confidence** - By testing the software, developers and stakeholders can have confidence that the software meets the requirements and works as intended.
- **Facilitates maintenance** - By identifying and fixing defects early, testing makes it easier to maintain and update the software.
- **Reduces costs** - Finding and fixing defects early in the development process is less expensive than fixing them later in the life cycle.

Disadvantages of Software Testing

1. **Time-consuming** - Testing can take a significant amount of time, particularly if thorough testing is performed.

1. **Resource-intensive** - Testing requires specialized skills and resources, which can be expensive.

1. **Limited coverage** - Testing can only reveal defects that are present in the test cases, and it is possible for defects to be missed.

1. **Unpredictable results** - The outcome of testing is not always predictable, and defects can be hard to replicate and fix.

1. **Delays in delivery** - Testing can delay the delivery of the software if testing takes longer than expected or if significant defects are identified.

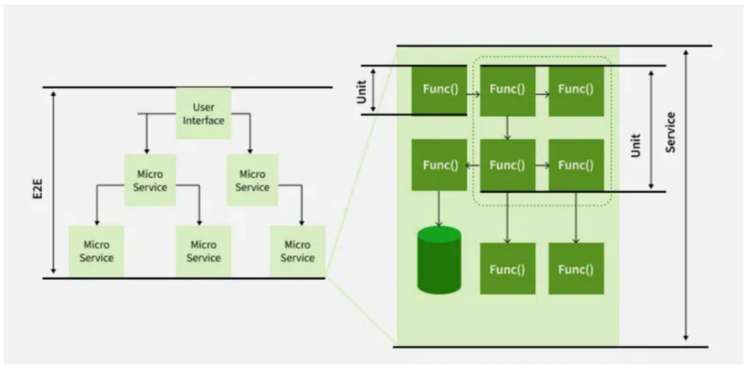
Unit Testing - Software Testing

Unit testing is the process of testing the smallest parts of your code, like it is a method in which we verify the code's correctness by running one by one. It's a key part of software development that improves code quality by testing each unit in isolation.

You write unit tests for these code units and run them automatically every time you make changes. If a test fails, it helps you quickly find and fix the issue. Unit testing promotes modular code, ensures better test coverage, and saves time by allowing developers to focus more on coding than manual testing.

What is a Unit Test?

A unit test is a small piece of code that checks if a specific function or method in is an application works correctly. It will work as the function inputs and verifying the outputs. These tests check that the code work as expected based on the logic the developer intended.



In these multiple tests are written for a single function to cover different possible scenarios and these are called test cases. While it is ideal to cover all expected behaviors, it is not always necessary to test every scenario.

Unit tests should run one by one, it means that they do not depend on other system parts like databases or networks. Instead, data stubs can be used to simulate these dependencies. Writing unit tests is easiest for simple, self-contained code blocks.

Unit testing strategies

To create effective unit tests, follow these basic techniques to ensure all scenarios are covered:

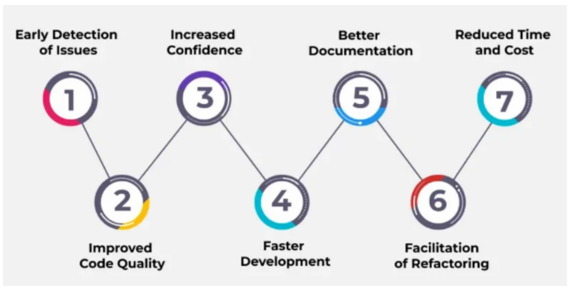
- **Logic checks**: Verify if the system performs correct calculations and follows the expected path with valid inputs. Check all possible paths through the code are tested.
- **Boundary checks**: Test how the system handles typical, edge case, and invalid inputs. For example, if an integer between 3 and 7 is expected, check how the system reacts to a 5 (normal), a 3 (edge case), and a 9 (invalid input).
- **Error handling**: Check the system properly handles errors. Does it prompt for a new input, or does it crash when something goes wrong?
- **Object-oriented checks**: If the code modifies objects, confirm that the object's state is correctly updated after running the code

Benefits of Unit Testing

Here are the Unit testing benefits which used in the software development with many ways:

- **Early Detection of Issues:** Unit testing allows developers to detect and fix issues early in the development process before they become larger and more difficult to fix.
- **Improved Code Quality:** Unit testing helps to ensure that each unit of code works as intended and meets the requirements, improving the overall quality of the software.

- **Increased Confidence:** Unit testing provides developers with confidence in their code, as they can validate that each unit of the software is functioning as expected.



- **Faster Development:** Unit testing enables developers to work faster and more efficiently, as they can validate changes to the code without having to wait for the full system to be tested.
- **Better Documentation:** Unit testing provides clear and concise documentation of the code and its behavior, making it easier for other developers to understand and maintain the software.
- **Facilitation of Refactoring:** Unit testing enables developers to safely make changes to the code, as they can validate that their changes do not break existing functionality.
- **Reduced Time and Cost:** Unit testing can reduce the time and cost required for later testing, as it helps to identify and fix issues early in the development process.

Disadvantages of Unit Testing

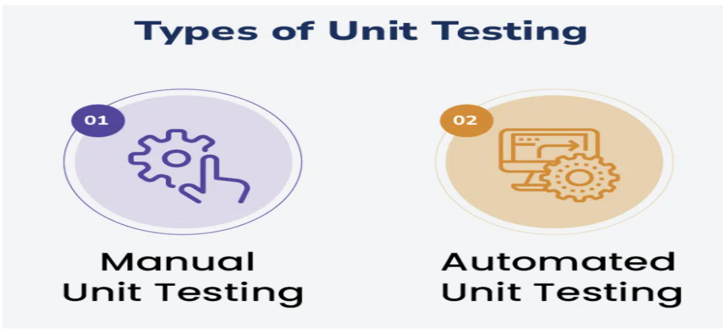
Here are the Unit testing dis-advantages which are follows:

- **Time and Effort:** Unit testing requires a significant investment of time and effort to create and maintain the test cases, especially for complex systems.
- **Dependence on Developers:** The success of unit testing depends on the developers, who must write clear, concise, and comprehensive test cases to validate the code.
- **Difficulty in Testing Complex Units:** Unit testing can be challenging when dealing with complex units, as it can be difficult to isolate and test individual units in isolation from the rest of the system.
- **Difficulty in Testing Interactions:** Unit testing may not be sufficient for testing interactions between units, as it only focuses on individual units.
- **Difficulty in Testing User Interfaces:** Unit testing may not be suitable for testing user interfaces, as it typically focuses on the functionality of individual units.
- **Over-reliance on Automation:** Over-reliance on automated unit tests can lead to a false sense of security, as automated tests may not uncover all possible issues or bugs.
- **Maintenance Overhead:** Unit testing requires ongoing maintenance and updates, as the code and test cases must be kept up-to-date with changes to the software.

Types of Unit Testing

Unit testing can be performed manually or automatically:

1. Manual unit testing



Manual Testing is like checking each part of a project by hand, without using any special tools. People, like developers, do each step of the testing themselves. But manual unit testing isn't used much because there are better ways to do it and it has some problems:

- It costs more because workers have to be paid for the time they spend testing, especially if they're not permanent staff.
- It takes a lot of time because tests have to be done every time the code changes.
- It is hard to find and fix problems because it is tricky to test each part separately.
- Developers often do manual testing themselves to see if their code works correctly.

Types of Unit Testing

2. Automated unit testing

Automation Unit Testing is a way of checking if software works correctly without needing lots of human effort. We use special tools made by people to run these tests automatically. These are part of the process of building the software. Here's how it works:

- Developers write a small piece of code to test a function in the software. This code is like a little experiment to see if everything works as it should.
- Before the software is finished and sent out to users, these test codes are taken out. They're only there to make sure everything is working properly during development.
- Automated testing can help us check each part of the software on its own. This helps us find out if one part depends too much on another. It's like putting each piece of a puzzle under a magnifying glass to see if they fit together just right.
- We usually use special tools or frameworks to do this testing automatically. These tools can tell us if any of our code doesn't pass the tests we set up.
- The tests we write should be really small and focus on one specific thing at a time. They should also run on the computer's memory and not need internet connection.

Unit test Procedure:

1. **Understand the unit and its requirements:**
    - Identify the specific "unit" of code to be tested, which is typically a function, method, or class.
    - Analyze the code to determine the different scenarios and potential test cases.

- Plan the inputs and expected outputs for each scenario.

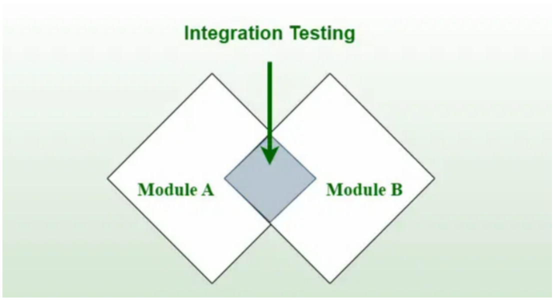
2. **Set up the testing environment:**
   - Choose and install a unit testing framework for your language (e.g., JUnit for Java, Jest for JavaScript, Pytest for Python).
   - Create a new project or add a new test project to your solution.
   - Add a reference from your test project to the project containing the code you want to test.
   - Consider using mocks and stubs to isolate the unit from its dependencies.
3. **Write the test cases using the AAA pattern:**
   - **Arrange:** Set up the objects and data needed for the test. This might involve creating instances of classes or setting up mock objects.
   - **Act:** Call the method or function you are testing with the arranged data.
   - **Assert:** Use assertion methods provided by the testing framework to check if the result of the "Act" step matches the expected outcome. For example, check if a value is equal to a specific number or if an error was thrown.
4. **Run the tests:**
   - Execute the tests using the framework's runner, which can often be done directly from your IDE (like Visual Studio) or via a command line.
   - Observe the results. Tests will either pass or fail.
5. **Interpret results and refactor:**
   - If a test fails, debug the code to find and fix the issue. Use the failing test to guide your debugging.
   - If all tests pass, you can proceed with refactoring the code with confidence, knowing the tests will catch any regressions.
   - Consider testing edge cases, invalid inputs, and potential failure points to make your tests more robust.

Integration Testing

Integration Testing is a Software Testing Technique that focuses on verifying the interactions and data exchange between different components or modules of a Software Application. The goal of Integration Testing is to identify any problems or bugs that arise when different components are combined and interact with each other.

- It mainly tests interface between two software units or modules. It focuses on determining the correctness of the interface. Once all the modules have been unit-tested, integration testing is performed.

- Integration testing can be done by picking module by module. This can be done so that there is a proper sequence to be followed.
- Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units.



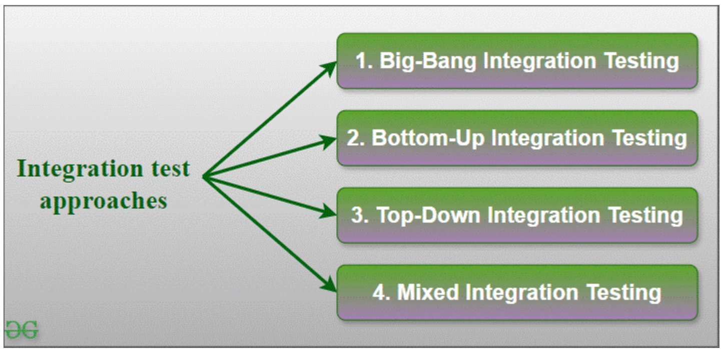
How to Write Integration Tests?

Designing integration test cases is a key part of ensuring that the different components of your software work well together. Here's a simplified approach to designing these tests:

- **Identify the components to be tested**: Start by pinpointing which parts of your software need to be tested together. These are usually modules that interact or depend on each other.
- **Determine the test objectives**: Define what you want to achieve with the test. Are you testing if data flows correctly between modules? Or perhaps checking if the system behaves as expected when components interact?
- **Define the test data**: Decide what data you'll use to test the integration. Make sure the data represents real-world scenarios so that your tests are relevant and meaningful.
- **Design the test cases**: Plan out the specific steps for each test. Think about what actions the test will take and what results you expect.
- **Develop test scripts**: Write the code that will automate your tests. If your tests are manual, ensure the steps are clearly documented and easy to follow.
- **Set up the testing environment**: Make sure the environment where the tests will run mimics the real-world setup as closely as possible. This will give you more accurate results.
- **Execute the tests**: Run your tests, paying close attention to how the components interact and whether they perform as expected.
- **Evaluate the results**: Finally, review the test outcomes. Did the components work as intended? Were there any errors or unexpected behaviors?

Types of Integration Testing

There are four main strategies for executing integration testing: big-bang, top-down, bottom-up, and sandwich (or hybrid) testing. Each of these methods comes with its own set of advantages and challenges, so it's important to choose the right one based on the specific needs of your project. Those approaches are the following:


Integration test approaches
1. Big-Bang Integration Testing
2. Bottom-Up Integration Testing
3. Top-Down Integration Testing
4. Mixed Integration Testing

Integration Test Approaches

1. Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules are combined and the functionality is verified after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated.

- In debugging errors reported during Big Bang integration testing is very expensive to fix.
- Big-bang integration testing is a software testing approach in which all components or modules of a software application are combined and tested at once.
- This approach is typically used when the software components have a low degree of interdependence or when there are constraints in the development environment that prevent testing individual components.
- The goal of big-bang integration testing is to verify the overall functionality of the system and to identify any integration problems that arise when the components are combined.
- While big-bang integration testing can be useful in some situations, it can also be a high-risk approach, as the complexity of the system and the number of interactions between components can make it difficult to identify and diagnose problems.

Advantages of Big-Bang Integration Testing

- It is convenient for small systems.
- Simple and straightforward approach.
- Can be completed quickly.
- Does not require a lot of planning or coordination.
- May be suitable for small systems or projects with a low degree of interdependence between components.

Disadvantages of Big-Bang Integration Testing

- There will be quite a lot of delay because you would have to wait for all the modules to be integrated.

- High-risk critical modules are not isolated and tested on priority since all modules are tested at once.
- Not Good for long projects.
- High risk of integration problems that are difficult to identify and diagnose.
- This can result in long and complex debugging and troubleshooting efforts.
- This can lead to system downtime and increased development costs.
- May not provide enough visibility into the interactions and data exchange between components.
- This can result in a lack of confidence in the system's stability and reliability.
- This can lead to decreased efficiency and productivity.
- This may result in a lack of confidence in the development team.
- This can lead to system failure and decreased user satisfaction.

2. Bottom-Up Integration Testing

In bottom-up testing, each module at lower levels are tested with higher modules until all modules are tested. The primary purpose of this integration testing is that each subsystem tests the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules.

Advantages of Bottom-Up Integration Testing

- In bottom-up testing, no stubs are required.
- A principal advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- It is easy to create the test conditions.
- Best for applications that uses bottom up design approach.
- It is Easy to observe the test results.

Disadvantages of Bottom-Up Integration Testing

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystems.
- As Far modules have been created, there is no working model can be represented.

3. Top-Down Integration Testing

Top-down integration testing technique is used in order to simulate the behaviour of the lower-level modules that are not yet integrated. In this integration testing, testing takes place from top to bottom. First, high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

Advantages of Top-Down Integration Testing

- Separately debugged module.
- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.
- Easier isolation of interface errors.
- In this, design defects can be found in the early stages.

Disadvantages of Top-Down Integration Testing

- Needs many Stubs.
- Modules at lower level are tested inadequately.
- It is difficult to observe the test output.
- It is difficult to stub design.

4. Mixed Integration Testing

A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. It is also called the hybrid integration testing. also, stubs and drivers are used in mixed integration testing.

Advantages of Mixed Integration Testing

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.
- Parallel test can be performed in top and bottom layer tests.

Disadvantages of Mixed Integration Testing

- For mixed integration testing, it requires very high cost because one part has a Top-down approach while another part has a bottom-up approach.
- This integration testing cannot be used for smaller systems with huge interdependence between different modules.

Applications of Integration Testing

Integration testing is all about making sure that different parts of a software application work well together. While unit testing checks individual components, integration testing focuses on how those components interact with each other. Here are the few application of Integration Testing:

1. **Identify the components:** Identify the individual components of your application that need to be integrated. This could include the frontend, backend, database, and any third-party services.

1. **Create a test plan:** Develop a test plan that outlines the scenarios and test cases that need to be executed to validate the integration points between the different components. This could include testing data flow, communication protocols, and error handling.

1. **Set up test environment:** Set up a test environment that mirrors the production environment as closely as possible. This will help ensure that the results of your integration tests are accurate and reliable.

1. **Execute the tests:** Execute the tests outlined in your test plan, starting with the most critical and complex scenarios. Be sure to log any defects or issues that you encounter during testing.

1. **Analyze the results:** Analyze the results of your integration tests to identify any defects or issues that need to be addressed. This may involve working with developers to fix bugs or make changes to the application architecture.

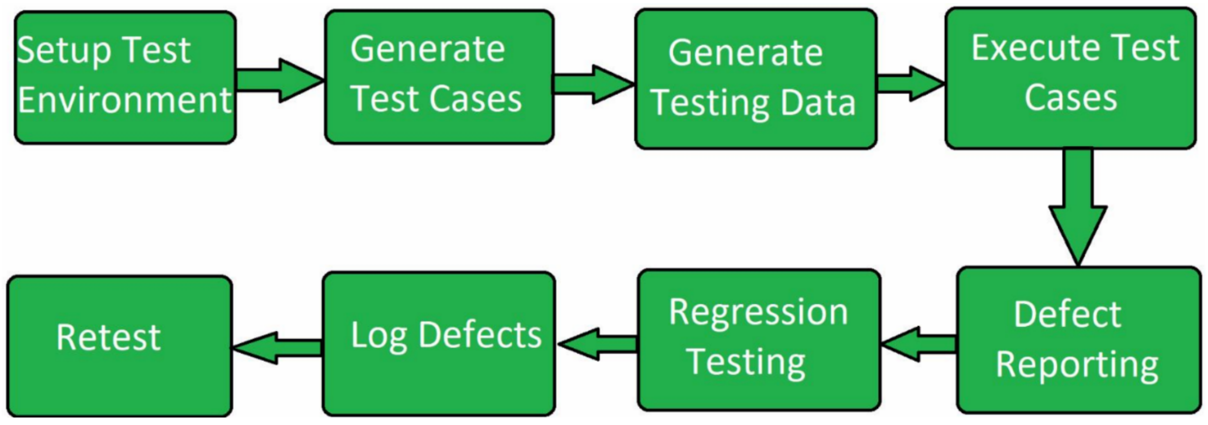
1. **Repeat testing:** Once defects have been fixed, repeat the integration testing process to ensure that the changes have been successful and that the application still works as expected.

System Testing

System Testing Process

System Testing is performed in the following steps:

- **Test Environment Setup:** Create testing environment for the better quality testing.
- **Create Test Case:** Generate test case for the testing process.
- **Create Test Data:** Generate the data that is to be tested.
- **Execute Test Case:** After the generation of the test case and the test data, test cases are executed.
- **Defect Reporting:** Defects in the system are detected.
- **Regression Testing:** It is carried out to test the side effects of the testing process.
- **Log Defects:** Defects are fixed in this step.
- **Retest:** If the test is not successful then again test is performed.



Types of System Testing

Here are the Types of System Testing are follows:

- **Functional Testing**: This checks if the system's features work as expected and meet the defined requirements.
- **Performance Testing**: This tests how the system performs under different conditions, like high traffic or heavy use, to ensure it can handle the expected load.
- **Security Testing**: This ensures the system's security measures protect sensitive data from unauthorized access or attacks.
- **Compatibility Testing**: This makes sure the system works well across different hardware, software, and network environments.
- **Usability Testing**: This evaluates how easy and user-friendly the system is, making sure it provides a good experience for users.
- **Regression Testing**: This ensures that any new code or features don't break or negatively affect the system's existing functionality.
- **Acceptance Testing**: This tests the system at a high level to make sure it meets customer expectations and requirements before release.

Advantages of System Testing

Here are the Advantages of System Testing are follows:

- In System Testing The testers do not require more knowledge of programming to carry out this testing.
- It will test the entire product or software so that we will easily detect the errors or defects which cannot be identified during the unit testing and integration testing.
- The testing environment is similar to that of the real time production or business environment.
- It checks the entire functionality of the system with different test scripts and also it covers the technical and business requirements of clients.

- After this testing, the product will almost cover all the possible bugs or errors and hence the development team will confidently go ahead with acceptance testing
- Verifies the overall functionality of the system.
- Detects and identifies system-level problems early in the development cycle.
- Helps to validate the requirements and ensure the system meets the user needs.
- Improves system reliability and quality.
- Facilitates collaboration and communication between development and testing teams.
- Enhances the overall performance of the system.
- Increases user confidence and reduces risks.
- Facilitates early detection and resolution of bugs and defects.
- Supports the identification of system-level dependencies and inter-module interactions.
- Improves the system's maintainability and scalability.

Disadvantages of System Testing

Here are the Disadvantages of System Testing are follows:

- System Testing is time consuming process than another testing techniques since it checks the entire product or software.
- The cost for the testing will be high since it covers the testing of entire software.
- It needs good debugging tool otherwise the hidden errors will not be found.
- Can be time-consuming and expensive.
- Requires adequate resources and infrastructure.
- Can be complex and challenging, especially for large and complex systems.
- Dependent on the quality of requirements and design documents.
- Limited visibility into the internal workings of the system.
- Can be impacted by external factors like hardware and network configurations.
- Requires proper planning, coordination, and execution.
- Can be impacted by changes made during development.
- Requires specialized skills and expertise.
- May require multiple test cycles to achieve desired results.

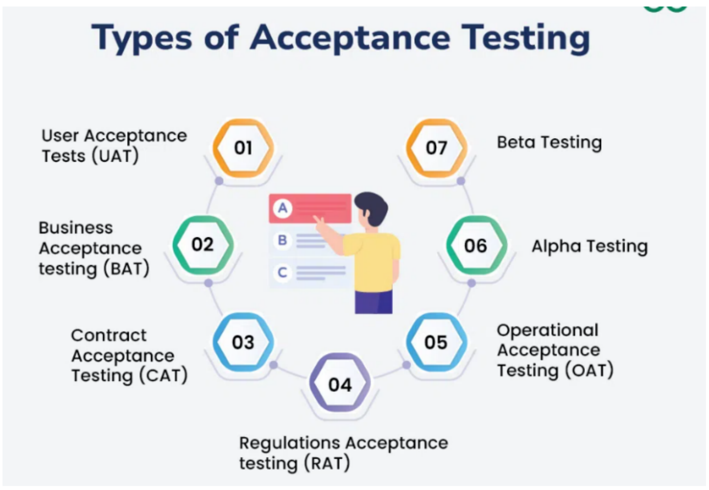
Acceptance Testing

Acceptance Testing is a formal testing according to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria or not and to enable the users, customers, or other authorized entities to determine whether to accept the system or not.

Types of Acceptance Testing

Here are the Types of Acceptance Testing

1. User Acceptance Testing (UAT)
1. Business Acceptance Testing (BAT)
1. Contract Acceptance Testing (CAT)
1. Regulations Acceptance Testing (RAT)
1. Operational Acceptance Testing (OAT)
1. Alpha Testing
1. Beta Testing


Types of Acceptance Testing
01
User Acceptance Tests (UAT)
02
Business Acceptance testing (BAT)
03
Contract Acceptance Testing (CAT)
04
Regulations Acceptance testing (RAT)
05
Operational Acceptance Testing (OAT)
06
Alpha Testing
07
Beta Testing
A
B
C

1. User Acceptance Testing (UAT)

- User acceptance testing is used to determine whether the product is working for the user correctly.
- Specific requirements which are quite often used by the customers are primarily picked for testing purposes. This is also termed as End-User Testing.

2. Business Acceptance Testing (BAT)

- BAT is used to determine whether the product meets the business goals and purposes or not.
- BAT mainly focuses on business profits which are quite challenging due to the changing market conditions and new technologies, so the current implementation may have to being changed which results in extra budgets.

3. Contract Acceptance Testing (CAT)

- CAT is a contract that specifies that once the product goes live, within a predetermined period, the acceptance test must be performed, and it should pass all the acceptance use cases.
- Here is a contract termed a Service Level Agreement (SLA), which includes the terms where the payment will be made only if the Product services are in-line with all the requirements, which means the contract is fulfilled.
- Sometimes, this contract happens before the product goes live.
- There should be a well-defined contract in terms of the period of testing, areas of testing, conditions on issues encountered at later stages, payments, etc.

4. Regulations Acceptance Testing (RAT)

- RAT is used to determine whether the product violates the rules and regulations that are defined by the government of the country where it is being released.
- This may be unintentional but will impact negatively on the business. Generally, the product or application that is to be released in the market, has to go under RAT, as different countries or regions have different rules and regulations defined by its governing bodies.
- If any rules and regulations are violated for any country then that country or the specific region then the product will not be released in that country or region.
- If the product is released even though there is a violation then only the vendors of the product will be directly responsible.

5. Operational Acceptance Testing (OAT)

- OAT is used to determine the operational readiness of the product and is non-functional testing.
- It mainly includes testing of recovery, compatibility, maintainability, reliability, etc. OAT assures the stability of the product before it is released to production.

6. Alpha Testing

- Alpha testing is used to determine the product in the development testing environment by a specialized testers team usually called alpha testers.

7. Beta Testing

- Beta testing is used to assess the product by exposing it to the real end-users, typically called beta testers in their environment.
- Feedback is collected from the users and the defects are fixed. Also, this helps in enhancing the product to give a rich user experience.

Use of Acceptance Testing

1. To find the defects missed during the functional testing phase.
1. How well the product is developed.
1. A product is what actually the customers need.

1. Feedback help in improving the product performance and user experience.
1. Minimize or eliminate the issues arising from the production.

Advantages of Acceptance Testing

1. This testing helps the project team to know the further requirements from the users directly as it involves the users for testing.
1. Automated test execution.
1. It brings confidence and satisfaction to the clients as they are directly involved in the testing process.
1. It is easier for the user to describe their requirement.
1. It covers only the Black-Box testing process and hence the entire functionality of the product will be tested.

Disadvantages of Acceptance Testing

1. Users should have basic knowledge about the product or application.
1. Sometimes, users don't want to participate in the testing process.
1. The feedback for the testing takes a long time as it involves many users and the opinions may differ from one user to another user.
1. Development team is not participated in this testing process.

Software Metrics

A metric is a measurement of the level at which any impute belongs to a system product or process.

Software metrics are a quantifiable or countable assessment of the attributes of a software product. There are 4 functions related to software metrics:

1. **Planning**
1. **Organizing**
1. **Controlling**
1. **Improving**

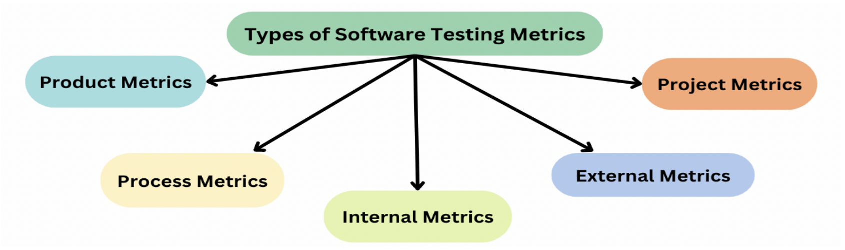
   Software metrics are **numbers** that help developers understand whether their software is **good, fast, clean, and correct**.

Characteristics of software Metrics

1. **Quantitative:** Metrics must possess a quantitative nature. It means metrics can be expressed in numerical values.
1. **Understandable:** Metric computation should be easily understood, and the method of computing metrics should be clearly defined.
1. **Applicability:** Metrics should be applicable in the initial phases of the development of the software.
1. **Repeatable:** When measured repeatedly, the metric values should be the same and consistent.
1. **Economical:** The computation of metrics should be economical.
1. **Language Independent:** Metrics should not depend on any programming language.

Types of Software Testing Metrics

Broadly, Software Metrics can be classified into the following types:



1. **Product Metrics**– Product Metrics quantify the features of a software product. First, the size and complexity of the product, and second, the dependability and quality of the software are the primary features that are emphasized.
2. **Process Metrics**– Unlike Product metrics, process metrics assess the characteristics of software development processes. Multiple factors can be emphasized, such as identifying defects or errors efficiently. In addition to fault detection, it emphasizes techniques, methods, tools, and overall software process reliability.
3. **Internal Metrics**– Using Internal Metrics, all properties crucial to the software developer are measured. Line of Control, or LOC, is an example of an internal metric.
4. **External Metrics**– Utilising External Metrics, all user-important properties are measured.
5. **Project Metrics**– The project managers use this metric system to monitor the project’s progress. Utilizing past project references to generate data. Time, cost, labor, etc., are among the most important measurement factors.

Types of Software Metrics

1▢Product Metrics

Tell us about the **final software product**.
Examples:

- Lines of Code (LOC)

- Number of bugs/defects
- Size of the software (modules, classes)
- Complexity (Cyclomatic complexity)

✔ *These metrics measure what the software is.*

2▢Process Metrics

Tell us about the **process used to build software**.
Examples:

- Development time
- Testing time
- Defect removal rate
- Number of change requests

✔ *Measures how well the development process works.*

3▢Project Metrics

These measure the overall project performance.

Examples:

- Cost
- Effort
- Productivity
- Team performance
- Resource allocation
- Risk metrics

Used for:

- Project planning
- Monitoring
- Controlling development activities
  - **Internal:** Metrics that can be measured directly from the software product itself, such as LOC.
  - **External:** Metrics that measure how the software behaves in its environment, often from a user's perspective, such as functionality and reliability.

Developing Software Metrics:

Developing software metrics involves identifying goals for improvement (e.g., quality, speed), selecting relevant metrics like cycle time, deployment frequency, and defect density, and establishing a system for consistent data collection and analysis. It is crucial to use these metrics for data-driven decisions, track trends, and focus on team-level insights rather than individual performance.

1. Identify goals and select metrics

- **Define objectives:** Start by identifying specific areas you want to improve, such as code quality, productivity, or delivery time.
- **Choose relevant metrics:** Select metrics that directly align with your goals. Some common types include:
  - **Productivity and workflow:** Team velocity, cycle time, lead time, and pull request review time.
  - **Quality and reliability:** Defect density, change failure rate, application crash rate, and code coverage.
  - **Performance:** Application performance metrics and mean time to recover (MTTR).

    - **Customer satisfaction:** Usability and Net Promoter Score (eNPS).

2. Implement data collection

- **Integrate tools:** Connect your code repositories, project management tools, and CI/CD pipelines to automate data collection and minimize manual effort.
  CI/CD (Continuous Integration / Continuous Delivery) pipelines allow organizations to **automatically collect, process, validate, and store data** without repeated manual work.
- These pipelines run **automated steps** whenever new data arrives, code changes, or scheduled triggers occur.
- **Gather data consistently:** Ensure that the metrics are computable, consistent, and obtained cost-effectively.

3. Analyze and act on the data

- **Establish benchmarks:** Set targets based on historical data or industry standards to provide a baseline for comparison.
- **Look for trends:** Analyze metrics for patterns and significant changes over time. Trends are often more insightful than isolated data points.
- **Use metrics for context:** Discuss the metrics during team retrospectives and check-ins to foster a collaborative approach to improvement.
- **Focus on team-level insights:** Use the metrics to identify bottlenecks and areas for improvement within the development process, rather than for individual scorekeeping.

Advantages of Software Metrics

1. Reduction in cost or budget.
1. It helps to identify the particular area for improvising.
1. It helps to increase the product quality.
1. Managing the workloads and teams.
1. Reduction in overall time to produce the product,.
1. It helps to determine the complexity of the code and to test the code with resources.
1. It helps in providing effective planning, controlling and managing of the entire product.

Disadvantages of Software Metrics

1. It is expensive and difficult to implement the metrics in some cases.
1. Performance of the entire team or an individual from the team can't be determined. Only the performance of the product is determined.
1. Sometimes the quality of the product is not met with the expectation.
1. It leads to measure the unwanted data which is wastage of time.
1. Measuring the incorrect data leads to make wrong decision making.

Complexity Software Metrics

Complexity metrics measure **how complicated the software design, module structure, or logic** is. High complexity usually means more errors, more testing effort, and harder maintenance. These metrics help developers identify risky or difficult parts of the system.

Below are the main complexity metrics along with simple numerical examples.

1▢Cyclomatic Complexity (McCabe's Metric)

Cyclomatic Complexity measures the number of **independent paths** in a module.
More decisions (if, loops, case) = more complexity.

Formula:

Cyclomatic Complexity (CC) = Number of decision points + 1

Calculation Example:

Suppose a module has:

- 2 IF statements
- 1 WHILE loop

Total decisions = 3

CC = 3 + 1 = 4

Meaning:

There are **4 independent paths** → high testing effort.

2⃣Halstead Complexity Metrics

Halstead metrics are based on **counting operators and operands** in a program.
Halstead Complexity Metrics were introduced by **Maurice Halstead** to measure the **size, complexity, and effort** required to develop or understand a program. These metrics are based purely on **counting operators and operands** in the source code.

1. Volume (V) – *"How big is the program?"*
Volume tells you **how large or complex the code is**, based on the number of operators and operands.

Think of it like measuring the **size of a book** based on the number of words it contains.

Higher Volume = bigger program = more to understand.

2. Difficulty (D) – *"How hard is the code to understand?"*
Difficulty tells you **how difficult the program is to read or understand**, based on:

- how many types of operators are used
- how many types of operands are used
- how many times operands appear

It measures the **mental effort required to understand or modify the code**.

Higher Difficulty = code is harder to read, understand, or debug.

3. Effort (E) – *"How much work is needed?"*

Simple meaning:
Effort tells you **how much total mental work** a programmer needs to:

- write the program
- understand it
- maintain it
- debug it

It is calculated using **Volume × Difficulty**, so it reflects both:

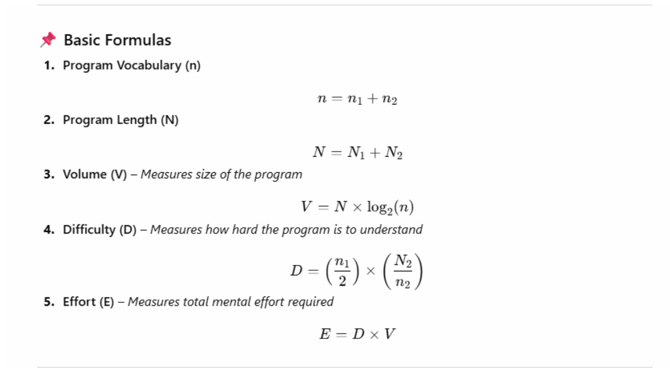
- the **size** of the program (Volume)
- and how **hard** it is to understand (Difficulty)

Higher Effort = more time and energy needed to work with the code.

Terminology:

- **n1** = number of distinct operators
- **n2** = number of distinct operands
- **N1** = total operators
- **N2** = total operands

Formulas:



📌 Basic Formulas

1. **Program Vocabulary (n)**

$$n = n_1 + n_2$$

2. **Program Length (N)**

$$N = N_1 + N_2$$

3. **Volume (V)** – *Measures size of the program*

$$V = N \times \log_2(n)$$

4. **Difficulty (D)** – *Measures how hard the program is to understand*

$$D = \left(\frac{n_1}{2}\right) \times \left(\frac{N_2}{n_2}\right)$$

5. **Effort (E)** – *Measures total mental effort required*

$$E = D \times V$$

Calculation Example:

Given:

- n1 = 3 distinct operators
- n2 = 4 distinct operands
- N1 = 10 operator occurrences
- N2 = 12 operand occurrences

Now calculate:

Program Vocabulary

n = n1 + n2 = 3 + 4 = 7

Program Length

N = N1 + N2 = 10 + 12 = 22

Volume

V = 22 × log2(7)

log2(7) ≈ 2.81

V = 22 × 2.81 ≈ 61.82

Difficulty

D = (n1 / 2) × (N2 / n2)

D = (3/2) × (12/4) = 1.5 × 3 = 4.5

Effort

E = D × V = 4.5 × 61.82 ≈ 278.19

3️⃣Structural Complexity Metrics (Coupling & Cohesion)

A. Coupling

Theory:

Coupling measures how many other modules a module is connected to.
More connections → more complexity.

Calculation Example:

Module A interacts with:

- Module B → 3 times
- Module C → 2 times
- Module D → 1 time

Total interactions:

Coupling = 3 + 2 + 1 = 6

High coupling (6) → high complexity.

B. Cohesion

Cohesion tells how closely related the internal functions of a module are.
Higher cohesion → lower complexity.

Calculation Example:

Module contains **6 functions**, out of which **3 perform related tasks**.

Cohesion = Related functions / Total functions

Cohesion = 3 / 6 = 0.5 (50%)

Lower cohesion → higher complexity.

4️⃣Size-Based Complexity Metric (Lines of Code – LOC)

Theory:

More lines of code → more complexity.
Does not measure quality but indicates effort.

Calculation Example:

Module has:

- 60 logical statements
- 15 comment lines (ignored)

LOC = 60

Higher LOC → more complexity.

5️⃣Object-Oriented Complexity Metrics (OOP)
Object-Oriented Complexity Metrics are used to measure the quality, complexity, and maintainability of object-oriented software.
These metrics evaluate features like classes, inheritance, objects, coupling, cohesion, and message passing**.**

A. WMC (Weighted Methods per Class)

Definition:
WMC counts the number of methods in a class. If each method has a complexity value (like cyclomatic complexity), WMC is the sum of complexities of all methods.

Meaning:

- More methods → class is more complex
- High WMC → harder to test, maintain, understand

Calculation Example:

Method complexities:

- Method1 = 4
- Method2 = 3
- Method3 = 5

WMC = 4 + 3 + 5 = 12

Higher WMC → complex class.

B. DIT (Depth of Inheritance Tree)

Definition:

DIT measures how many levels a class is down the inheritance hierarchy.

Meaning:

- High DIT → more inherited behavior → more complexity
- Deeper trees → harder to understand and predict behavior

Calculation Example:

Class E inherits from D,
D → C,
C → B,
B → A.

Levels = 4

DIT = 4

Greater DIT → more complexity.

C. CBO (Coupling Between Objects)

Definition:
CBO measures how many other classes a class is directly connected to.

Meaning:

- High coupling → class depends on many others → difficult to modify
- Low coupling → better modularity and maintainability

Calculation Example:

Class X uses:

- Class M
- Class N
- Class O

CBO = 3

Higher CBO → more complexity.

D. RFC (Response for Class)

Definition:
RFC counts the number of methods that can be executed in response to a message sent to an object.
(Own methods + methods called inside those methods)

Meaning:

- High RFC → class has more behavior → more complex
- More possible responses → harder to test

Calculation Example:

Class contains:

- 5 methods of its own
- 4 methods from other classes it can call

RFC = 5 + 4 = 9

Higher RFC → more complex behavior.

Software Quality Assurance (SQA)

Quality Concepts

Quality in software means **how well the software meets customer needs, works correctly, and performs reliably**.

✔ **Key Quality Concepts:**

Software Quality Assurance

Topics included:

1. **Quality Concepts**
2. **Quality Movement**
3. **Background Issues**
4. **SQA Activities**
5. **Formal Approaches to SQA**
6. **Statistical Quality Assurance**
7. **Software Reliability**

1 Quality Concepts

Software quality means how well the software meets user needs and performs error-free.

✔ **Key ideas:**

SOFTWARE QUALITY

Software Quality means *how well a software system meets customer requirements, performs its functions without errors, and provides a good user experience*. High-quality software is reliable, efficient, secure, and easy to use.

Explanation in Detail

1. **Meeting User Requirements**
   - Software must do what the customer expects.
   - Requirements may be functional (features) or non-functional (speed, security).
   - If the delivered software does not match customer needs, it is considered low quality.
2. **Correctness**
   - Software should produce accurate results.
   - No calculation mistakes, logical errors, or incorrect outputs.
3. **Reliability**
   - Software must work without failing during normal use.
   - If it crashes frequently, it is considered unreliable.
4. **Efficiency**
   - Uses system resources (CPU, memory, storage) effectively.
   - Should not slow down the system.
5. **User Satisfaction**
   - Software should be easy to learn and operate.
   - The better the experience, the higher the quality.
6. **Standards Compliance**
   - Software should follow recognized quality and coding standards like ISO or IEEE.
   - Ensures consistency and maintainability.

Why Software Quality is Important

- Reduces maintenance cost.
- Improves user trust.
- Prevents failures during use.
- Enhances business reputation.

QUALITY CONTROL (QC)

Quality Control is a *product-oriented* activity. It focuses on identifying defects **after** the software is developed.

Explanation in Detail

1. **Purpose of QC**
   - To detect defects in the finished software product.
   - Ensures the final application is error-free and ready for release.
2. **Activities in QC**
   - **Testing** (unit test, system test, integration test)
   - **Reviewing results**
   - **Bug tracking**
   - **Re-testing and regression testing**
3. **QC is Reactive**
   - Means it reacts to problems.
   - Defects are found after they occur.
4. **Focus on Deliverables**
   - QC checks the correctness of:
     - Code
     - GUI
     - Database
     - Functions
     - Performance
   - The output must meet requirements.
5. **Tools Used**
   - Selenium, JUnit, LoadRunner, QTP (UFT), JIRA, Bugzilla.
6. **Outcome of QC**
   - Assurance that the product has minimal defects.
   - Ensures the software is stable and ready for customer use.

Example

If a login page accepts wrong credentials without validation, QC testers will detect this issue and report it as a defect.

QUALITY ASSURANCE (QA)

Quality Assurance is a *process-oriented* activity. It focuses on **preventing defects** by improving and monitoring the development process.

Explanation in Detail

1. **Purpose of QA**
   - To ensure a good development process so that defects do not occur in the first place.
   - QA ensures the right steps are followed during development.
2. **QA is Proactive**
   - QA prevents problems before they appear.
   - Activities include audits, training, process definition, and standards implementation.
3. **Process-Oriented**
   - QA focuses on:
     - Development processes
     - Testing processes
     - Documentation practices
     - Coding standards
   - Ensures the team follows the best methods.
4. **QA Activities**
   - Process audits
   - Code review guidelines
   - Test process improvement
   - Quality planning
   - Root cause analysis (RCA)
5. **Tools Used**
   - Test management tools, requirement management tools, auditing tools.
6. **Outcome of QA**
   - Reduced defects in the final product.
   - Consistent and high-quality development cycle.
   - Continuous improvement of organizational processes.

Example

QA team ensures there is a proper review process for code, which reduces bugs before testing starts.

QUALITY STANDARDS

Quality standards are **documented guidelines and frameworks** used to ensure that software development follows best practices and maintains consistent quality.

Explanation in Detail

1. **Purpose**
    - Provide a structured way to develop, test, and maintain software.
    - Ensures uniform quality in every project.
2. **Popular Software Quality Standards**
    - **ISO 9001**
        - International standard for Quality Management Systems (QMS).
        - Ensures customer satisfaction and continuous improvement.
    - **IEEE Standards**
        - Provide guidelines for software processes:
            - IEEE 829 (Test Documentation)
            - IEEE 830 (Requirements Specification)
    - **CMMI (Capability Maturity Model Integration)**
        - Measures the maturity of software processes.
        - Levels range from Initial (Level 1) to Optimizing (Level 5).
3. **Importance of Standards**
    - Reduces project failure.
    - Ensures predictable outcomes.
    - Improves communication between teams.
    - Helps in certification and global recognition.
4. **How Standards Improve Quality**
    - Define clear rules for development.
    - Reduce risks and errors.
    - Improve consistency in coding and testing.
5. **Documentation with Standards**
    - Requirements document
    - Test plan
    - Design document
    - User manual

Conclusion

Quality standards ensure that software is delivered with expected quality, meeting customer needs and industry benchmarks.

QUALITY ATTRIBUTES

Quality attributes are **characteristics that define how good the software is**, beyond just basic functionality.

Below is a detailed explanation of each attribute:

1. Functionality

- Software must correctly perform all required functions.
- Includes features, accuracy, and security.
- Incorrect functionality = system failure.

2. Reliability

- Ability of software to work without failure for a long time.
- Includes:
    - Fault tolerance (works even when errors occur)
    - Recoverability (how fast it recovers after crash)

3. Usability

- How easily a user can operate the software.

- Factors:
    - Simple UI
    - Good navigation
    - Clear instructions
- Good usability = better user satisfaction.

4. Maintainability

- How easily software can be updated or fixed.
- Includes:
    - Understandable code
    - Modular design
    - Proper documentation
- High maintainability reduces cost of future changes.

5. Performance

- Speed and responsiveness of the software.
- Includes:
    - Response time
    - Throughput
    - Efficient use of memory and CPU
- High performance = smooth user experience.

6. Security

- Protects data from unauthorized access.
- Includes:
    - Authentication
    - Authorization
    - Encryption
    - Data integrity
- Very important in banking, healthcare, e-commerce.

7. Portability

- Ability of software to run on different environments.
- Example:
    - Works on Windows, Linux, and macOS.
    - Runs on mobile and desktop.

2 Quality Movement

The Quality Movement refers to how the concept of quality evolved over time—from basic inspection to modern process-based quality assurance. It shows how industries (including software engineering) gradually improved their approach to achieving high quality.

Key Contributors to the Quality Movement

1. **Deming** – Continuous improvement, PDCA cycle (Plan–Do–Check–Act)
2. **Juran** – Quality planning and management
3. **Crosby** – Zero defects, "Quality is free"
4. **Ishikawa** – Cause-and-effect diagrams, quality circles
5. ISO Quality Standards

   Provide international guidance for quality management.

Why the movement matters for software?

- Software must be reliable, fast, secure, and error-free.
    - ✓ **Core Principles of the Quality Movement in SQA**

The quality movement in SQA is built on principles adapted from general industrial quality management (pioneered by figures like Walter Shewhart, W. Edwards Deming, and Joseph Juran) and tailored for the unique challenges of software development.

- Process-Oriented Approach: A central tenet is that the quality of a product is highly influenced by the quality of the process used to acquire, develop, and maintain it. SQA focuses on monitoring and improving these processes, rather than just the final product.
- Defect Prevention: The emphasis moved from reactive quality control (finding and fixing bugs after they occur) to proactive quality assurance (preventing defects from occurring in the first place). This includes activities like formal technical reviews, code reviews, and early requirement analysis.
- Customer Satisfaction: Quality is ultimately defined by the customer's needs and expectations. The movement stresses understanding and meeting these expectations, which go beyond just basic functionality to include non-functional requirements like usability, performance, and security.
- Continuous Improvement: SQA is not a one-time activity but an ongoing cycle of planning, doing, checking, and acting (PDCA cycle) to refine processes and enhance software quality over time.
- Standards and Compliance: The adoption of internationally recognized standards, such as the ISO 9000 family and models like the Capability Maturity Model Integration (CMMI), provides frameworks and benchmarks for organizations to measure and improve their quality processes.

✓ **Key Influences and Modern Practices**

The quality movement has led to the adoption of several key practices in modern SQA:

- Shift-Left Testing: This practice involves moving testing activities earlier in the SDLC to detect and address issues when they are less costly to fix.
- Agile and DevOps Integration: Quality assurance is integrated into every sprint and stage of the continuous integration/continuous deployment (CI/CD) pipelines, fostering constant collaboration between development and testing teams.
- Automated Testing: The use of tools like Selenium and cloud-based platforms enhances testing efficiency and supports continuous testing within CI/CD pipelines.
- Metrics and Data-Driven Decisions: SQA relies on collecting and analysing data on defect density, test coverage, and customer satisfaction to make informed decisions about process improvements.

3 Background Issues in Software Quality

Background issues are the hidden factors that influence software quality throughout the development lifecycle. These issues make it difficult to achieve error-free, reliable, and maintainable software.

1. Software Complexity

Software becomes complex as it grows in size and features.

Why This Is a Problem

- Modern software contains **thousands of lines of code**, many functions, classes, and interconnected modules.
- As complexity increases, **interactions between modules become harder to predict**.
- One small change in one module can break another part of the system.

Effects on Quality

- Increased chance of bugs and logical errors.
- Difficult to understand and maintain the system.
- Harder to test all combinations and paths.

Example

A banking system with many modules (login, funds transfer, statements, loans, security, etc.) is highly complex and difficult to test completely.

2. Changing Customer Requirements

Customers may change requirements frequently due to business needs or new market trends.

Why This Is a Problem

- Developers start building one requirement, but the customer later demands changes.
- Frequent modifications may break the existing design.
- Improper documentation of new changes leads to misunderstandings.

Effects on Quality

- Leads to incomplete or inconsistent features.
- Testing becomes difficult because test cases must be updated again and again.
- Software may be delivered with defects due to rushed changes.

Example

In an e-commerce project, the client may change payment gateway requirements frequently, causing code rework and new bugs.

3. Lack of Communication

Miscommunication between stakeholders affects software accuracy.

Why This Happens

- Poor interaction between developers, testers, analysts, and customers.
- Requirements not explained clearly.
- Missing or misunderstood information.

Effects on Quality

- Wrong features get developed.
- Testers may test incorrect functionality.
- Leads to rework, delays, and defects.

Example

A customer wants a “search by date” feature, but developers build a “search by month” option due to misunderstanding.

4. Time and Budget Pressure

Projects often run on tight deadlines or limited budgets.

Why This Is a Problem

- Teams rush to complete tasks.
- Less time is available for proper coding, review, and testing.

Effects on Quality

- Insufficient testing leads to more bugs in production.
- Developers skip best practices and proper documentation.
- Results in unstable, unreliable software.

Example

Releasing software before a festival sale (Amazon/Flipkart) may force the team to deliver incomplete features.

5. Lack of Standards

Quality standards define how software should be developed, coded, and tested.

Why This Is a Problem

When no standards are followed:

- Developers write code in different styles.

- Testing is done inconsistently.
- No uniform process is followed for documentation or reviews.

Effects on Quality

- Hard to maintain or understand the code later.
- High chances of defects due to no coding rules.
- New team members struggle to follow the process.

Examples of Missing Standards

- No naming conventions
- No test documentation guidelines
- No coding guidelines

6. Technology Challenges

Software development must keep up with new tools, platforms, and frameworks.

Why This Is a Problem

- Developers may lack experience with new technologies.
- Tools may contain bugs or limitations.
- Compatibility issues arise between different versions of technology.

Effects on Quality

- More errors because developers do not fully understand new tools.
- Difficulty in integration and testing.
- Higher risk of system failures.

Example

Migrating from Java 8 to Java 17 may cause compatibility issues unless handled carefully.

7. Poor Documentation

Documentation includes requirement documents, design documents, test cases, and user manuals.

Why This Is a Problem

- Without proper documentation, developers and testers cannot understand what needs to be done.
- Hard to maintain or update software in the future.

Effects on Quality

- Misinterpretation of requirements.
- Difficult for new team members to understand the system.
- Poor testing coverage.

Example

If the requirement specification is unclear, the team might build features that the customer did not want.

8. Human Resource Issues

Quality depends on the skills and experience of the team.

Problems

- Lack of training

- Inexperienced developers
- Frequent employee turnover

Effect on Quality

- Increases mistakes and rework.
- Leads to poor design, coding errors, and missed test cases.

9. Inadequate Testing Tools and Infrastructure

Software testing needs proper tools, hardware, and environments.

Problems

- Lack of automation tools
- Slow or outdated systems
- No proper test environment (like production)

Effects on Quality

- Inefficient testing
- Missing critical bugs
- Poor performance and reliability

10. Poor Project Management

Project managers play a major role in quality.

Problems

- Improper planning
- Poor scheduling
- Inadequate risk management

Effects on Quality

- Scope creep
- Uncontrolled changes
- Delayed delivery
- Reduced focus on quality activities

11. Integration Issues

Large systems are divided into modules developed by different teams.

Problems

- Modules may not integrate smoothy
- Interface mismatches
- Dependency issues

Effects on Quality

- System failures during integration
- Unexpected behavior

4 SQA Activities

SQA Activities are the set of tasks carried out to ensure that the software development process follows quality standards, reduces defects, and produces a reliable product.
These activities focus on **prevention of defects**, **process improvement**, and **ensuring quality throughout the SDLC**.

Below are the major SQA activities

SQA Activities are the set of tasks carried out to ensure that the software development process follows quality standards, reduces defects, and produces a reliable product.
These activities focus on **prevention of defects**, **process improvement**, and **ensuring quality throughout the SDLC**.

Below are the major SQA activities

1. Requirements Analysis and Review

SQA ensures that the requirements gathered from the customer are:

- Complete
- Clear
- Consistent
- Testable

Why important?

- Incorrect or unclear requirements lead to major defects later.
- Saves time and cost by catching mistakes early.

Activities

- Conduct requirement reviews.
- Verify feasibility.
- Ensure all requirements are properly documented.

2. Preparing and Following Standards

SQA ensures the use of coding standards, design standards, testing standards, and documentation standards.

Why important?

- Standards bring consistency.
- Makes software easier to maintain.
- Reduces defects caused by poor coding style or documentation.

Examples

- IEEE documentation standards
- ISO 9001 standards
- Organization-specific coding rules

3. Quality Planning

Quality planning defines how quality will be achieved for a project.

Activities

- Preparing the SQA plan.
- Defining quality goals.
- Selecting tools, techniques, and metrics.
- Identifying risks and mitigation strategies.

Why important?

- Creates a roadmap for achieving the required quality level.

4. Conducting Reviews (Peer Reviews / Code Reviews)

Reviews help in identifying defects early in the development stage.

Types of Reviews

- Peer reviews
- Code reviews
- Design reviews
- Management reviews
- Walkthroughs
- Inspections

Benefits

- Cheaper than fixing defects during testing.
- Helps maintain coding standards.
- Improves overall quality and team learning.

5. Software Testing

Although QA focuses on processes, testing is still a major activity under SQA.

Types of Testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Performance testing
- Automation testing

Purpose

- To detect defects in the software.
- To ensure the system meets customer requirements.

6. Process Monitoring and Improvement

SQA checks whether the development process is being followed properly.

Activities

- Monitoring each phase of SDLC.
- Conducting audits (process audits, technical audits).
- Identifying weaknesses in the development process.
- Suggesting improvements (Corrective and Preventive Actions).

Purpose

- Ensures processes remain effective and error-free.
- Encourages continuous improvement (like PDCA cycle).

7. Configuration Management

Configuration Management (CM) ensures that any change in the software is properly controlled.

Activities

- Version control of code and documents.
- Tracking changes (who changed what and when?).
- Maintaining baseline versions.

Tools

- Git
- SVN
- GitHub / GitLab

Purpose

- Prevents loss of code.
- Ensures changes do not break existing features.

8. Audits (Quality Audits & Process Audits)

Audits are formal evaluations to check if the team is following the required process or standards.

Types

- Internal audits
- External audits

Purpose

- To verify compliance with standards (ISO, CMMI).
- Identify deviations and recommend corrective actions.

9. Defect Management & Root Cause Analysis (RCA)

SQA ensures defects are properly recorded, analyzed, and resolved.

Activities

- Logging defects.
- Classifying severity and priority.
- Performing RCA to prevent future defects.
- Tracking defect trends using metrics.

Purpose

- Helps in reducing recurring errors.
- Improves product and process quality.

10. Training and Knowledge Sharing

SQA ensures team members are trained in:

- Standards
- Tools
- Best practices
- Testing techniques
- Security practices

Purpose

- Skilled teams produce fewer defects.
- Keeps everyone updated with latest technologies.

11. Risk Management

SQA helps identify and mitigate risks that may affect quality.

Examples of risks

- Technology failure
- Skill shortage
- Schedule delays

12. Metrics Collection and Analysis

SQA collects data to measure and improve quality.

Examples of metrics

- Defect density
- Test coverage
- Productivity metrics
- Mean Time to Failure (MTTF)

5 Formal Approaches to SQA

Formal approaches are mathematically–based techniques used to ensure software correctness, reliability, and quality.
Unlike testing or reviews, formal methods use mathematical models, logic, proofs, and formal specifications to verify that software behaves exactly as expected.

These approaches help detect errors early—especially in critical systems like banking, aviation, defense, medical devices, and industrial automation.

1. Formal Specification

Formal specification means describing software requirements using mathematical notation instead of natural language.

Why is this needed?

- Natural language (English) often creates ambiguities.
- Mathematics removes confusion—everything is exact.

Characteristics

- Based on set theory, functions, logic.
- Precise, unambiguous requirements.
- Helps developers clearly understand what system must do.

Benefits

- Detects missing, unclear, or conflicting requirements.
- Improves communication between teams.
- Reduces defects later in design and coding.

Example

Using mathematical predicates to define behavior of a login function (like conditions for valid/invalid user input).

2. Formal Verification

Formal verification uses mathematical proof techniques to check if the system meets its specification.

Methods Used

- Theorem proving
- Model checking
- Proof obligations
- Automated verification tools

Purpose

- Ensures software behaves correctly in all cases.
- Used when failure can cause huge losses (e.g., aircraft autopilot software).

Benefits

- Proves correctness logically.
- Detects errors that testing may never find.

3. Model Checking

Model checking is an automated technique that verifies whether a software model satisfies a given property or rule.

How it works

- System is represented as a “state model”.
- Tool explores **all possible states** to detect violations.

Used For

- Deadlock detection in concurrent systems
- Checking safety and security properties

Benefits

- Fully automatic
- Detects hidden, rare errors (e.g., memory deadlocks)
- Used widely in safety-critical software.

4. Formal Reviews and Inspections

These are structured and systematic review processes performed using formal rules.

Includes

- Code inspections
- Design inspections
- Requirements inspections
- Walkthroughs with checklists

Purpose

- Identify defects early through human review.
- Ensure compliance with standards and specifications.

Benefits

- Very cost-effective
- Removes 60–70% of defects before testing stage

Why "formal"?

- Uses predefined rules, entry/exit criteria, checklist, review roles (moderator, author, reviewer)

5. Formal Testing Techniques

Testing is made more scientific using formal methods.

Examples

- Boundary value analysis
- Equivalence class partitioning
- Control flow testing
- Data flow testing
- Mathematically created test cases

Benefits

- Ensures maximum coverage
- Reduces redundancy
- Improves defect detection rate

This makes testing systematic, measurable, and predictable.

6. Mathematical Modeling

Uses mathematical models to represent behavior of software.

Techniques

- State transition diagrams
- Petri nets
- Finite automata
- Decision tables

Purpose

- Understand complex systems
- Detect logical issues
- Identify unreachable states or infinite loops

7. Proof of Correctness

A mathematical proof that a program is correct for all inputs.

Includes

- Pre-conditions
- Post-conditions
- Loop invariants

Used When

- High-risk algorithms (encryption, safety-critical software).

8. Formal Configuration Management

Formal procedures for managing changes in software.

Activities

- Change request evaluation
- Impact analysis
- Version control rules

Ensures high quality and consistency even when the software evolves.

6 Statistical Quality Assurance

```
        +----------------------------+
        | Statistical Quality        |
        |      Assurance (SQA)       |
        +-------------+--------------+
                      |
   ---------------------------------------------
   |                  |               |           |
Data Collection   Data Analysis   Identify    Improve Process
                                  Patterns
```

Statistical Quality Assurance (SQA)

Statistical Quality Assurance means using statistical tools and techniques to measure, monitor, analyze and improve software quality throughout the development process.

Statistical Quality Assurance refers to using statistical techniques like control charts, defect density, Pareto analysis, and reliability models to measure and improve software quality. It helps detect variations, predict failures, identify high-defect modules, and make data-driven decisions.

Why Use Statistics in SQA?

Using statistics gives **quantitative**, **objective**, and **measurable** results instead of depending on guesswork.

✓ Key Benefits:

- Helps **detect trends** in defects.
- Allows prediction of **future quality**.
- Identifies **process variation**.
- Helps managers make **data-driven decisions**.
- Improves **reliability and customer satisfaction**.

1)**Defect Density**

Defect Density tells **how many defects** are present in a software module per unit size.

✓ Formula

Defect Density = Number of Defects / Size of Software (KLOC or Function Points)

✓ Why it is used?

- To identify **which module is of good or poor quality**.
- Lower defect density = **better quality**.

✓ Example Calculation

- Module A: 10 defects, Size = 2 KLOC
- Module B: 8 defects, Size = 1 KLOC

Compute:

Module A:

Defect Density = 10 / 2 = 5 defects/KLOC

Module B:

Defect Density = 8 / 1 = 8 defects/KLOC

✔ Even though Module A has more defects, Module B has a **higher defect density**, so it is **poorer in quality**.

2) Pareto Analysis (80/20 Rule)

Pareto principle says **80% of defects come from 20% of modules**.

This helps focus effort where it matters most.

✔ **Why use it?**

- To identify **high-risk, problem-making modules**.
- Helps in **prioritizing testing**.

✔ **Example**

Suppose defects are found in modules:

Module	Defects
Login	40
Payment	30
Cart	20
Search	5
Profile	5

Total defects = 100

Now calculate percentage:

- Login: 40%
- Payment: 30%
- Cart: 20%
- Search + Profile: 10%

Here, **Login + Payment + Cart (3 modules)** cause **90% defects**.

✔ These 3 modules are **vital few** (20%).
✔ Remaining modules are **trivial many**.

3)**Cause and Effect Diagram (Ishikawa / Fishbone)**

Identifies the **root cause** of defects rather than just fixing symptoms.

✔ **Common categories:**

- Methods
- Machines
- People
- Materials

- Measurement
- Environment

✔ **Example**

Defect: *Login fails randomly.*

Possible causes:

- **Methods:** Wrong validation logic
- **Machines:** Server CPU overload
- **People:** Tester missed boundary values
- **Materials:** Wrong version of config file
- **Measurement:** Logging not capturing error
- **Environment:** Low network bandwidth

✔ Helps understand **WHY defects happen**.

4) **Control Charts**

Control charts show whether the **software development or testing process is stable**.

✔ **Key Elements**

- **UCL** – Upper Control Limit
- **LCL** – Lower Control Limit
- **CL (mean)** – Average of data points

✔ **Example (Defects per build)**

Build	Defects
1	5
2	6
3	4
4	5
5	15

Now calculate:

- CL = (5+6+4+5+15)/5 = 7 defects (approx)
- UCL = CL + 3σ
- LCL = CL – 3σ
  (σ = standard deviation)

Here, Build 5 = **15**, which is **much higher** than others.

✔ Build 5 exceeds UCL → **process is out of control** (something unusual happened).
✔ Indicates **special cause of variation** such as:

- bad code merge
- missing test cases
- unstable build

5)**Scatter Diagram**

Shows **relationship between two variables**.

✔ **Example**

Let's compare **complexity** vs **defects**:

Complexity	Defects
10	2
20	4
30	6
40	9

As complexity increases, defects also increase.

✓ Shows **positive correlation**.
✓ Helps predict defects for future modules.

6)**Histogram (Defect Distribution)**

A histogram shows **how many times** each type of defect occurred.

✓ **Example**

Defects found:

- UI errors = 12
- Database errors = 8
- Logic errors = 5
- Performance errors = 3

Histogram shows:

- UI errors occur most
- Performance errors occur least

✓ Helps prioritize which area needs more attention.

7)**Statistical Sampling**

Instead of checking the **entire software**, we check a **sample** (like 20% of modules).

✓ **Why use?**

- Saves time
- Reduces cost
- Good when project is very large

✓ **Example**

Total modules = 50
Sample size = 20% = 10 modules

Defects in sample = 25 defects

Estimate total defects:

If 10 modules → 25 defects
Then 50 modules → (25 × 5) = 125 defects (estimated)

✓ Helps estimate **overall quality** without checking everything.

8)**Software Reliability Models (Statistical Models)**

These models predict:

- Remaining defects
- Expected failures

- Probability of failure-free use

✔ **Common models**

- Jelinski–Moranda
- Goel–Okumoto
- Musa's Execution Time model

✔ **Simple Example**

Suppose during testing, defects found per week are:

Week	**Defects Found**
1	20
2	12
3	8
4	5
5	3

Trend is **decreasing**.

Using any reliability model:
✔ You predict fewer defects will be found in coming weeks.
✔ Software reliability is **improving**.

-

7 Software Reliability

Software reliability means the probability that the software will perform without failure for a specific time under given conditions.

Software Reliability is the **probability that software will operate without failure** for a specified time under specified conditions.

✔ In simple words:
How likely is the software to run without crashing?

✔ It deals with:

- failures
- error occurrence
- defect removal
- system behavior in real usage

2. Why Software Reliability Is Important?

Because unreliable software leads to:

- frequent crashes
- inconsistent output
- customer dissatisfaction
- safety risks
- increased maintenance cost

Reliability ensures:

- stable performance
- fewer failures
- improved user trust
- lower support cost

3. Characteristics of Software Reliability

Software reliability depends on:

- **Number of remaining defects**
- **Operational environment** (load, users, inputs)
- **Execution time**
- **Quality of development and testing**

Unlike hardware, software:

- does **not wear out**
- failures occur due to **bugs**, not physical damage

4. Key Terms in Software Reliability

◆ **Failure**

When the software does not perform its expected function.
Example: App crashes when login button is clicked.

◆ **Fault (Bug)**

The cause of the failure — incorrect logic or code.

◆ **Error**

Human mistake during coding or design.

Error → Fault → Failure

5. Software Reliability Metrics

1) MTBF (Mean Time Between Failures)

MTBF = MTTF + MTTR

◆ **MTTF – Mean Time To Failure**

Time the system runs before the next failure.

◆ **MTTR – Mean Time To Repair**

Time taken to fix and restore service.

✔ **Example**

A system runs for 900 hours, and total repair time in that period is 100 hours.

Given:

- MTTF = 900 hours
- MTTR = 100 hours

MTBF = 900 + 100 = 1000 hours

Higher MTBF = higher reliability.

2) **Failure Rate (λ)**

Number of failures per unit time.

Failure Rate = Number of Failures / Total Time

✔ **Example**

10 failures in 500 hours:

λ = 10/500 = 0.02 failures per hour

Lower failure rate = more reliable.

3)**Reliability Function R(t)**

Probability that software runs without failure until time t.

R(t) = e ^ (-λt)

Example:
Failure rate λ = 0.01
Find reliability for 50 hours:

R(50) = e ^ (-0.01 × 50)
 = e ^ (-0.5)
 ≈ 0.607 (60.7%)

Meaning:
Software has **60% chance** of running failure-free for 50 hours.

6. Software Reliability Growth Curve

As testing continues:

- defects are found
- defects are fixed
- failures reduce over time

  Graph shape:

```
Failures
 ↑
 | *
 |  *
 |    *
 |      *
 |         *
 +----------------------→ Time
```

✔ Failures gradually **decrease**
✔ Reliability **increases**

This is called **Reliability Growth Model**.

7. Factors Affecting Software Reliability

◆ **1. Complexity**

Highly complex code → more failures.

◆ **2. Testing quality**

Better testing → fewer remaining defects.

◆ **3. Programming practices**

Consistent coding standards improve reliability.

◆ **4. Operational environment**

More load or unusual inputs reduce reliability.

🔹 **5. Change requests**

Frequent changes introduce new faults.

8**. Steps to Improve Software Reliability**

✔ **1. Better requirements**

Clear and stable requirements reduce defects.

✔ **2. Code reviews**

Find defects early.

✔ **3. Rigorous testing**

Unit, integration, regression, performance testing.

✔ **4. Defect prevention techniques**

Root cause analysis (RCA), design checklists.

✔ **5. Using reliability tools**

Control charts, fault tree analysis, statistical models.

9**. Real-life Example of Software Reliability**

Example: **Mobile Banking App**

- Expected uptime = 99.9%
- Failure must be <0.1%
- MTTR must be very low (few minutes)
- Reliability must remain high under load

If app crashes frequently → low reliability.