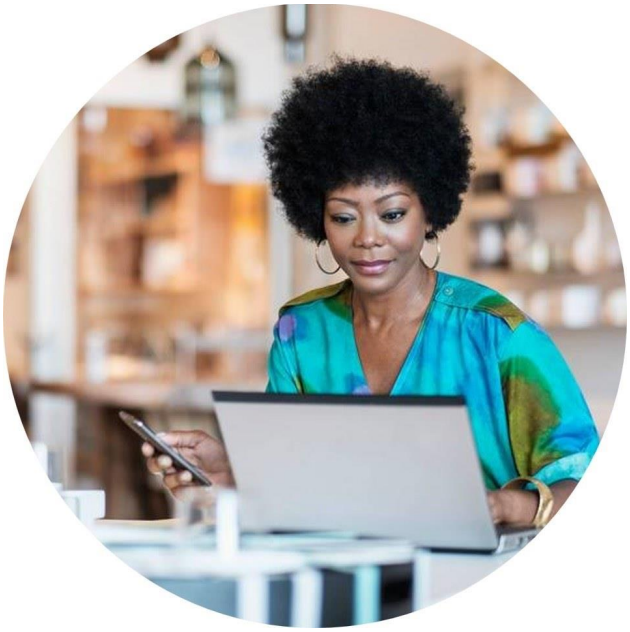
UNIT-2(STQA)

UNIT-2 - STQA

Software Testing and Quality Assurance UNIT - 2

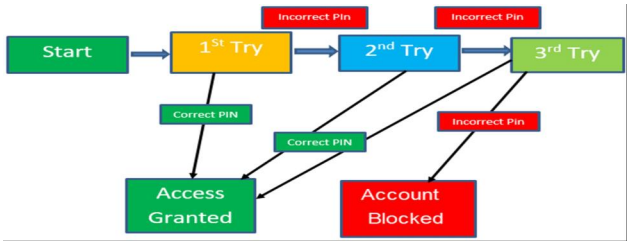


Agenda

- Transition Factor Testing Classes
- Black Box Testing
- Testing Process
- Test Case Design
- Automated Testing
- Testing Life Cycle
- Software Quality Implementation
- Quality of software maintenance components
- Quality of external Participants' contributions
- CASE tools and their effect on software Quality

BBT-Transition Factor Testing Classes

This type of testing is used when a system moves from one state to another (example: login screen → dashboard). It checks whether the transitions between states work properly.



1. Valid Transitions

These are the normal, expected transitions that the system should allow.

Example: When the correct username and password are entered, the user should go from the login page to the dashboard.

✓ This test ensures that correct behaviour is followed when valid inputs are given.

2. Invalid Transitions

These test cases check for unexpected or incorrect transitions that should not happen.

Example: Entering the wrong password should not allow access to the dashboard.

✓ This ensures that the system blocks invalid actions and handles them correctly (e.g., showing an error message).

3. Boundary Tests

These test transitions that happen at the edges or limits of the system's behavior.

Example: If the system allows only 3 login attempts, testing what happens at the 3rd and 4th attempt is a boundary test.

✓ Helps in catching bugs that occur at limits or extreme conditions.

4. State Coverage

This checks whether all possible states in the system are visited at least once during testing.

Example: In a shopping app, states might include browsing, cart, payment, and order confirmation.

✓ Ensures that no part of the system (state) is left untested.

5. Transition Coverage

This makes sure that every possible transition (movement from one state to another) is tested.

Example: From login → dashboard, dashboard → logout, dashboard → settings, etc.

✓ It verifies that the system behaves correctly when moving from one state to another in all directions.

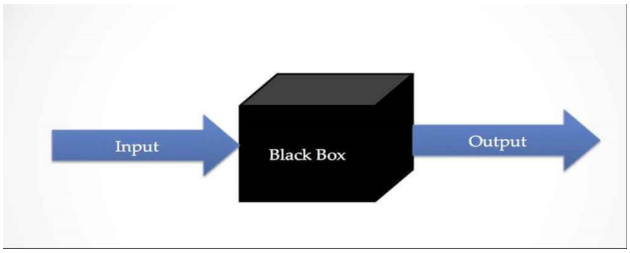
Black Box Testing

Black Box Testing is a software testing method where the tester does not need to know the internal code or logic of the program.

The focus is only on the inputs given and the outputs received — like testing a machine by pressing buttons without opening it.

Example:

You test a login screen by entering a username and password. If the login succeeds or fails as expected, the test passes — you don't care how the code works inside.



✅ Advantages of Black Box Testing

No Coding Knowledge Needed

Testers don’t need to know how the code is written — they just test based on requirements.

Can Be Done by Non-Developers

People like testers, domain experts, or users can perform it, making it flexible and cost-effective.

Great for Functional Testing

It helps check whether each feature (like login, search, checkout) works properly from the user’s or customer point of view.

❌ Disadvantages of Black Box Testing

Can't Detect Internal Errors

If there is a problem deep inside the code logic, this method might miss it because we’re not looking inside the code.

Some Parts of Code May Go Untested

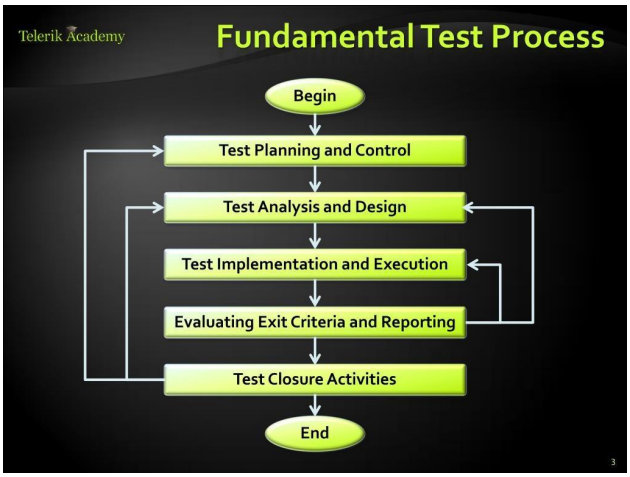
Since testers don’t know the code structure, they might miss some conditions or logic paths.

Difficult to Cover All Scenarios

Testers may not guess all combinations of inputs, especially in complex systems — so some bugs may be missed.

Testing Process

The Testing process is a series of structured steps followed to verify and validate a software product. It ensures that the software meets user requirements and is free from major defects before release.



Steps in the Testing Process:

1.Test Planning

- This is the first and most important step.
- Decide what to test, how to test, who will test, and when to test.
- Identify required tools, risks, schedule, and resources.
- Output: Test Plan Document

2. Test Design

- Prepare test cases based on the software's requirements.
- Each test case includes inputs, expected outputs, and test data.
- Use techniques like Boundary Value Analysis, Equivalence Partitioning, etc.
- Output: Test Case Document

3. Test Implementation (Test Development)

- Develop and organize manual or automated test scripts.
- Set up the test environment (tools, test data, software build).
- Ensure all test cases are reviewed and ready for execution.
- Output: Test Scripts and Prepared Environment

4. Test Execution

- Run the test cases using the inputs and check actual vs. expected results.

- Record test results as Pass, Fail, or Blocked.
- Automated or manual testing is performed based on project setup.
- Output: Test Execution Reports

5. Defect Reporting and Tracking

If a test fails, the tester logs the defect in a bug tracking tool with all necessary details like steps to reproduce, expected vs actual results, screenshots, and severity level (how serious the issue is). This helps developers clearly understand and fix the issue. After fixing, the tester retests to confirm the bug is resolved. This cycle continues until all major bugs are closed.

- Track each defect until it's fixed and retested.
- Tools used: Jira, Bugzilla, Mantis, etc.
- Output: Defect Logs and Reports

6.Test Closure

Once major defects are fixed and testing goals are achieved, testing is formally closed.

Prepare a Test Summary Report with metrics (total cases, pass/fail rate, open defects).

Document lessons learned and best practices for future use.

Output: Test Closure Report

Test Case Design

What is a Test Case?

A test case is a set of conditions or steps used to check whether a software feature works as expected.

Example:

Feature: Login Page

Input: Username = "abc", Password = "123"

Expected Result: Successful login

Purpose of Test Case Design

To create test cases that effectively check the correctness, completeness, and quality of the software with minimum effort and time.

Common Test Case Design Techniques

1. Equivalence Partitioning

- Divide inputs into valid and invalid groups (partitions).
- Example: For input 1–10 → Valid: 5, Invalid: -1, 11
- Only one value from each group is tested, saving time.
- Focus on testing at the edges of input ranges.

2. Boundary Value Analysis

- Example: For range 1–10 → Test: 0, 1, 10, 11
- Many errors happen at the boundary values.

3. State Transition Testing

Useful for software that changes states based on input.

Example: ATM – States: Card Inserted → PIN Entered → Transaction

Tests if valid transitions between states are working.

4. Error Guessing

Based on experience and intuition, guess what errors users might make.

Example: Leaving fields blank, entering special characters, etc.

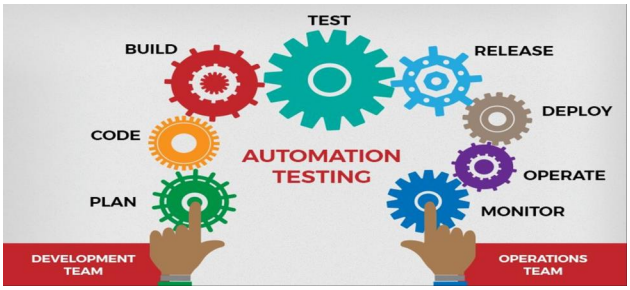
Test case - Example

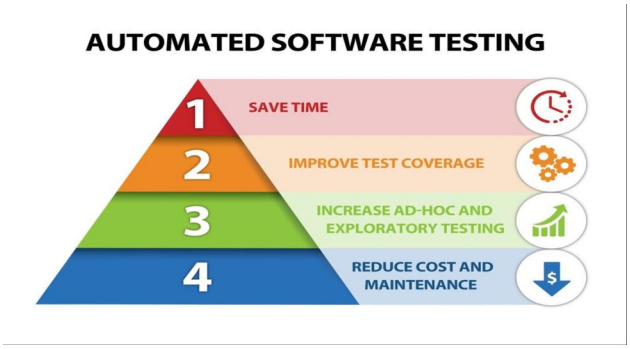
S.No	Action/Description	Input	Expected Result	Actual result	status	Comment
1	Enter 4 alphabets and click on create button	asdf	It should accept	It is accepting	Passed	nil
2	Enter 16 alphabets and click on create button	asdfghjklzxcvbnm	It should accept	It is accepting	Passed	
3	Enter special characters and click on create button	!@#$$$	It should not accept	It is not accepting	Passed	

Automated Testing

What is Automated Testing?

Automated testing is a process where testing tasks are performed automatically using tools, instead of being done manually by a person. Test scripts are written once and can be run multiple times without human effort.





Benefits of Automated Testing:

- Faster Execution: Saves time by running tests quickly.
- Repeatability: You can run the same tests again and again (useful for regression testing).
- High Accuracy: Reduces chances of human error.
- Good for Large Projects: Efficient when testing big or complex software.

Disadvantages of Automated Testing:

- High Initial Cost: Tools and setup can be expensive.
- Requires Skilled Testers: Needs people who can write and maintain automation scripts.
- Limited to Certain Tests: Not suitable for new features, UI testing, or tests that need human observation (like colour or visual checks).

Examples of Popular Automation Tools:

Tool	Used For
Selenium	Web application testing
JUnit	Unit testing in Java programs
QTP/UFT	Functional and regression testing

When to Use Automated Testing?

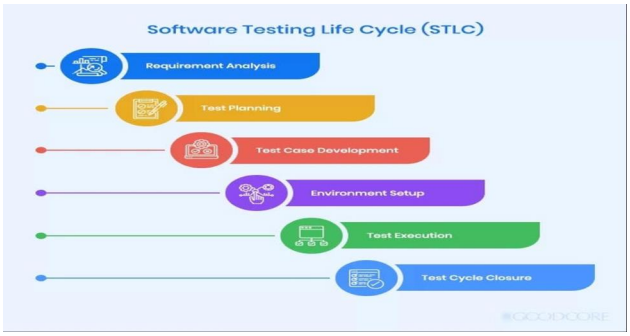
For **regression testing** (repeating old tests).

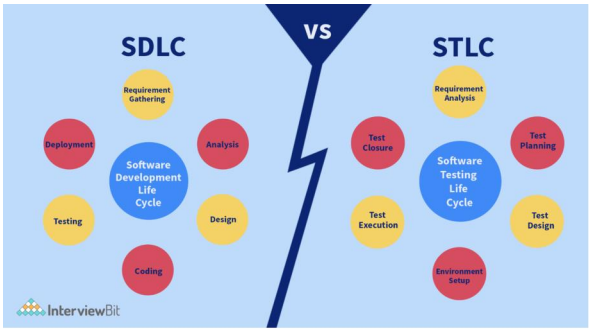
For **data-driven testing** (same test with many inputs).

For **load and performance testing**.

Testing Life Cycle (STLC)

Software Testing Life Cycle (STLC) is the sequence of activities performed during the testing process to ensure software quality. It starts with understanding requirements and ends with test closure and reporting. This is the life cycle followed in testing from start to finish.





Phases of STLC:

1. Requirement Analysis

- Understand what needs to be tested from the project requirements.
- Identify testable items, test objectives, and types of tests needed.
- Involve testers early to find any gaps or unclear areas in requirements.

2. Test Planning

- Decide how testing will be done, who will do it, and what tools will be used.
- Define the testing scope, schedule, budget, and risk management.

- Output: Test Plan Document.

3. Test Case Development

- Create test cases and test data based on the requirements.
- Define the expected results for each test case.
- Get test cases reviewed and approved.

4. Test Environment Setup

- Prepare the system where testing will happen.
- This includes hardware, software, network, and test tools.
- Ensure it's similar to the production environment for realistic results.

5. Test Execution

- Run the test cases and compare actual results with expected results.
- If mismatches (bugs) are found, log them and report to the development team.
- Track the status of each test (Pass/Fail).

6. Test Closure

- After completing all planned testing, prepare the Test Summary Report.
- Analyze test coverage, defect status, and testing metrics.
- Share lessons learned and improve for future projects.

Software Quality Implementation

Software Quality Implementation means ensuring the software is developed with quality from the beginning, not just tested at the end.

- It includes following coding standards, regularly reviewing code, using tools to check code quality and performance, training team members on best practices, and testing early in the development process.

These steps help prevent bugs and improve overall software reliability.

1. Follow Standard Coding Practices

- Developers should follow recognized coding guidelines (like Google Java Style Guide, PEP8, etc.) for consistent coding structure. This ensures code is readable, maintainable,

and less prone to bugs. It also helps new team members understand and contribute easily.

2. Conduct Regular Code Reviews

- Team members should regularly review each other's code before it goes live. Code reviews help in detecting logic issues, improving performance, and ensuring that coding standards are followed. They also promote collaboration and learning within the team.

3. Use Quality Tools

- Integrate tools like SonarQube for code analysis, JUnit for unit testing, and JMeter or LoadRunner for performance testing. These tools automatically highlight bugs, vulnerabilities, and poor code practices, saving time and improving code quality.

4. Train the Team on Quality

- Arrange training programs, workshops, and knowledge-sharing sessions on the latest quality tools and practices. Training helps developers and testers stay updated and aligned with quality objectives. It encourages a culture of continuous improvement.

5. Test Early and Often

- Start testing during development using methods like Unit Testing, TDD (Test-Driven Development), and Continuous Testing. Frequent and early testing helps identify and fix bugs before they grow into bigger issues. This improves the stability and reliability of the final software product.

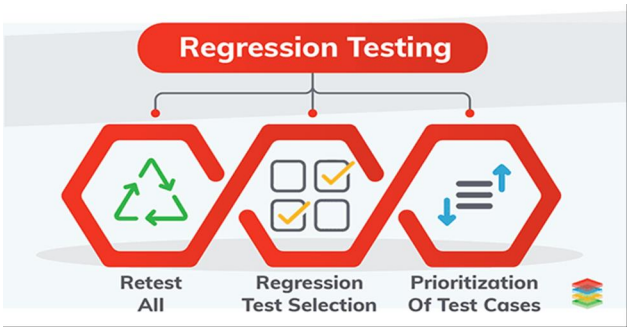
Assuring Quality in Maintenance Components

After software is delivered to the client or released to users, it often requires updates, bug fixes, security patches, or feature enhancements. This phase is known as software maintenance. Ensuring quality during maintenance is critical to keep the system stable, usable, and reliable.

Key Practices:

1. Perform Regression Testing

- Whenever a new change or fix is made, there’s a risk that it may unintentionally affect other parts of the software. Regression testing is done to re-run existing test cases to ensure previously working features still work correctly after changes. This prevents new updates from introducing hidden bugs.



2. Conduct Impact Analysis

- Before making any change, developers should analyse its impact on the rest of the system. Impact analysis helps in identifying what modules or components may be affected by the change. This ensures that only safe and controlled changes are made without disturbing other working parts.

3. Keep Documentation Updated

- Every change or fix should be properly documented — what was changed, why it was changed, and how it was tested. Updated documentation helps new developers understand the current state of the software and avoids confusion during future maintenance.

4. Track Bugs or Issues Properly

- Use bug tracking tools (like Jira, Bugzilla, etc.) to record, assign, and monitor issues. This makes sure no bug is missed, and helps in prioritizing and organizing the fixing process. Proper tracking also helps in generating reports and understanding recurring issues.

5. Why Quality in Maintenance is Important?

- It prevents new problems while fixing old ones.
- It keeps the software stable, secure, and user-friendly.
- It builds trust with users and reduces future support costs.
- After software is released, updates and fixes are needed. This is maintenance.

6. To ensure quality:

- Use regression testing (to check nothing else broke).
- Perform impact analysis before making changes.
- Keep documentation updated.

- Track bugs or issues properly.

Assuring Quality of External Contributors

In many software projects, external contributors are involved. These could be:

- Freelancers hired for short-term tasks
- Third-party companies/vendors working on modules
- Open-source developers contributing to shared projects

Since they are not part of the internal team, we must take extra steps to ensure their work is high quality and safe to use.

1. Set Clear Expectations

Before they begin, explain things clearly:

- What features or tasks they are responsible for
- What tools, languages, or frameworks to use
- When the work should be completed (deadlines)
- What level of quality, performance, and security is expected

📌 *Clear instructions avoid confusion, rework, and delays.*

2. Review Their Code

- Once the external contributor submits code:
- Internal team members must review it line by line
- Check for logic errors, security risks, and bad practices
- Make sure the code follows the company's coding standards

📌 *This ensures the code is clean, understandable, and safe.*

3. Test Before Integrating

- Never merge external code into the main system without testing:
- First, test the new code using unit tests (small parts)
- Then use integration tests (check how it works with other parts)
- Finally, check that it does not break existing features (regression testing)

📌 *Testing protects the whole system from unexpected problems.*

4. Use Common Tools and Standards

- To work smoothly with external people:
- Use version control like Git or GitHub
- Follow common code formatting rules
- Set up CI/CD tools (like Jenkins, GitLab CI) to run tests automatically

📌 *When everyone uses the same tools, it's easier to collaborate and avoid mistakes.*

Final Note:

- Working with external people is helpful, but it needs proper planning and checking. By setting rules, reviewing, and testing properly, we can ensure their work is reliable, secure, and fits well with the rest of the project.

CASE Tools and Software Quality

CASE (Computer-Aided Software Engineering) Tools are software applications that help developers, testers, and designers in different phases of software development. These tools automate many tasks like designing, coding, testing, and documentation.

Types of CASE Tools

1. Upper CASE Tools

- Used during early stages: requirements gathering, planning, and system design
- Help create UML diagrams, DFDs, ER diagrams
- Example: IBM Rational Rose, StarUML

2. Lower CASE Tools

- Used during later stages like coding, testing, and maintenance
- Help in code generation, debugging, unit testing
- Example: JUnit, Selenium

3. Integrated CASE Tools

- Combine both upper and lower CASE features
- Support the entire Software Development Life Cycle (SDLC)

- Example: Enterprise Architect, Visual Paradigm

Impact of CASE Tools on Software Quality

✅ Increases Productivity

  - Developers save time by automating repetitive tasks
  - Faster design, testing, and documentation

✅ Improves Consistency

  - Code style, design patterns, and formats remain uniform across the team

✅ Reduces Human Errors

  - Automatic checks and validations help detect mistakes early

✅ Improves Documentation and Maintainability

  - Auto-generated documents make it easy to update and maintain the system