Software Testing Lifecycle, Levels, Types, and Techniques

A Comprehensive Guide for QA and QE Professionals

INTRODUCTION

What This Course Covers

01

Test Lifecycle (STLC)

Understanding the complete software testing process from planning to closure

02

Test Levels

Exploring unit, integration, system, and acceptance testing layers

03

Test Types

Functional, non-functional, regression, and specialised testing approaches

04

Testing Techniques

Static vs dynamic and black box vs white box methodologies

CHAPTER 1

Software Testing Lifecycle (STLC)

The Software Testing Lifecycle (STLC) is a systematic sequence of activities conducted during the testing process. Unlike the Software Development Lifecycle (SDLC), STLC focuses exclusively on testing activities, ensuring quality assurance throughout development.

Each phase has specific entry criteria, activities, deliverables, and exit criteria. Understanding STLC helps testers plan, execute, and track testing efforts effectively whilst ensuring comprehensive test coverage.

The Six Phases of STLC

Requirement Analysis

Identify testable requirements and gather test requirements

Test Planning

Define test strategy, estimate effort, and allocate resources

Test Case Development

Create detailed test cases and prepare test data

Environment Setup

Configure test environment with necessary hardware and software

Test Execution

Execute test cases and log defects

Test Closure

Evaluate completion criteria and document learnings

STLC Phase Details: Entry and Exit Criteria

Phase	Entry Criteria	Exit Criteria
Requirement Analysis	Requirements document available, stakeholders identified	RTM created, automation feasibility assessed
Test Planning	Requirements analysed, effort estimated	Test plan approved, resources allocated
Test Case Development	Test plan approved, requirements understood	Test cases reviewed and approved
Environment Setup	Test plan available, environment requirements defined	Environment ready, smoke test passed
Test Execution	Test cases ready, test data prepared	All test cases executed, defects logged
Test Closure	Testing completed, exit criteria met	Test summary report published

INDUSTRY EXAMPLE

STLC in Practice: E-Commerce Platform

Scenario

A major e-commerce company launching a new payment gateway integration must follow STLC to ensure secure, reliable transactions.

Approach

- **Requirement Analysis:** Identified 45 testable requirements including PCI-DSS compliance
- **Test Planning:** 3-week timeline with 5 testers, risk-based prioritisation
- **Test Development:** Created 200+ test cases covering security and functionality
- **Environment Setup:** Configured sandbox with dummy payment APIs
- **Execution:** Found 23 defects, 3 critical security issues
- **Closure:** 100% defect resolution before production release

INDUSTRY EXAMPLE

STLC in Practice: Banking Mobile App

Context

Regional bank upgrading mobile app with biometric authentication and fund transfer features

Challenge

Tight 6-week release window, regulatory compliance requirements, multiple device compatibility

STLC Application

Parallel test planning and case development saved 1 week. Risk-based testing prioritised security features first.

Outcome

Delivered on time with 98% test coverage, zero production defects in first month

QUICK QUIZ

Test Your STLC Knowledge

Question 1

Which STLC phase involves creating the Requirements Traceability Matrix (RTM)?

- A) Test Planning
- B) Requirement Analysis
- C) Test Case Development
- D) Test Execution

Answer: B) Requirement Analysis

Question 2

What is the primary deliverable of the Test Closure phase?

- A) Test cases
- B) Defect reports
- C) Test summary report
- D) Test plan

Answer: C) Test summary report

Question 3

Which phase can begin in parallel with Test Planning to save time?

- A) Environment Setup
- B) Test Case Development
- C) Test Execution
- D) Requirement Analysis

Answer: B) Test Case Development

Interview Keywords: STLC

Entry/Exit Criteria

Conditions that must be met before starting or completing a phase

RTM

Requirements Traceability Matrix maps requirements to test cases

Risk-Based Testing

Prioritising testing efforts based on risk assessment

Test Metrics

Quantitative measures like defect density and test coverage

Smoke Testing

Preliminary testing to verify build stability

Test Deliverables

Artefacts produced during testing like test plans and reports

CHAPTER 2

Test Levels

Test levels represent different stages of testing that correspond to different stages of software development. Each level has specific objectives, scope, and responsibility.

The test pyramid concept suggests more tests at lower levels (unit) and fewer at higher levels (acceptance). This approach provides faster feedback, easier debugging, and cost-effective quality assurance.

Understanding test levels helps teams allocate testing resources efficiently and ensures comprehensive coverage across the application stack.

The Four Primary Test Levels

Unit Testing

Testing individual components or modules in isolation. Performed by developers using frameworks like JUnit or NUnit.

System Testing

Testing the complete integrated system against specified requirements. End-to-end testing of the entire application.

Integration Testing

Testing interfaces and interactions between integrated components or systems.

Acceptance Testing

Validating the system meets business requirements and is ready for deployment. Includes UAT and alpha/beta testing.

Test Levels: Detailed Comparison

Aspect	Unit	Integration	System	Acceptance
Scope	Single module	Module interfaces	Complete system	Business requirements
Performed By	Developers	Testers/Developers	QA Team	End users/Clients
Environment	Development	Test environment	Test environment	Production-like
Defect Cost	Very Low	Low	Medium	High
Focus	Code correctness	Interface defects	End-to-end flows	User satisfaction
Automation	High (90%+)	Medium (60-70%)	Medium (50-60%)	Low (30-40%)

DEEP DIVE

Integration Testing Approaches

Top-Down Integration

Testing starts from top-level modules, moving downwards. Uses stubs to simulate lower modules. Advantages: Early prototype possible. Disadvantages: Critical modules tested last.

Bottom-Up Integration

Testing begins with lowest modules, moving upwards. Uses drivers to simulate higher modules. Advantages: Critical modules tested first. Disadvantages: No early prototype.

Sandwich/Hybrid Integration

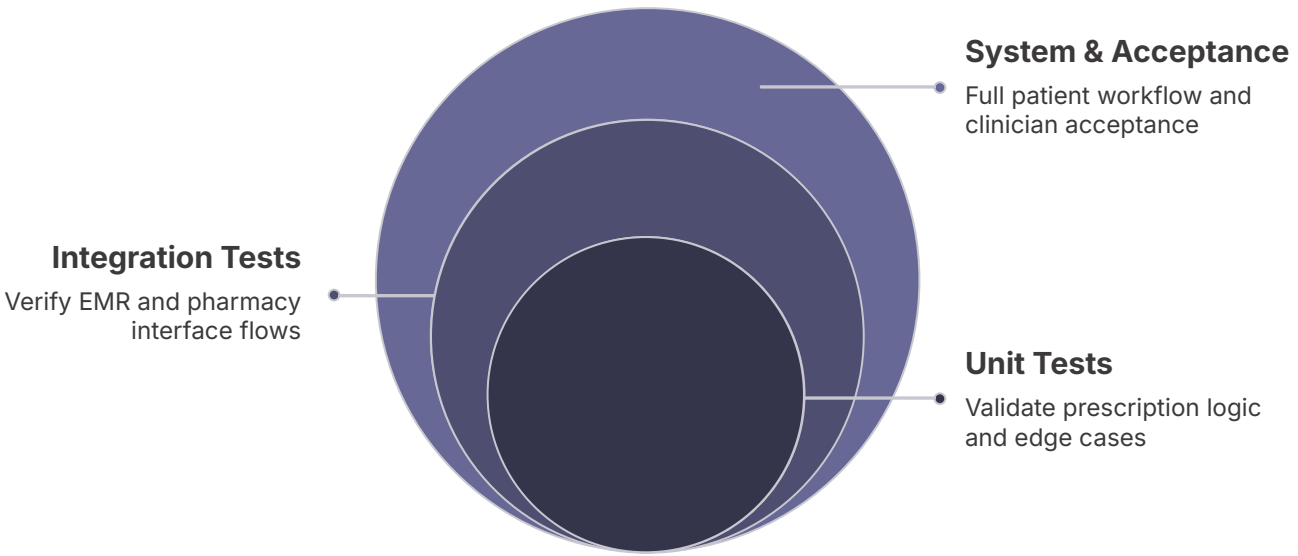
Combines top-down and bottom-up approaches simultaneously. Tests top and bottom layers in parallel. Advantages: Balanced approach, faster. Disadvantages: Requires more resources.

Big Bang Integration

All modules integrated and tested together at once. No incremental testing. Advantages: Simple approach. Disadvantages: Difficult defect localisation, high risk.

INDUSTRY EXAMPLE

Test Levels in Action: Healthcare System



Hospital Management System Implementation

A hospital implementing an integrated management system applied all test levels systematically:

- **Unit Level:** Developers wrote 1,200+ unit tests for prescription validation, dosage calculations, and patient record functions
- **Integration Level:** QA tested interfaces between EMR, laboratory, pharmacy, and billing systems using API testing
- **System Level:** End-to-end testing of patient admission, treatment, and discharge workflows
- **Acceptance Level:** Doctors and nurses validated clinical workflows in UAT environment

Result: Zero critical defects post-deployment, smooth go-live across 3 hospital locations.

INDUSTRY EXAMPLE

Test Levels: Streaming Service Platform

Video Streaming Service Launch

A new streaming platform applied test levels to ensure quality at scale. **Unit testing** covered video encoding algorithms and user authentication functions with 95% code coverage. **Integration testing** validated CDN integration, payment gateway connections, and recommendation engine APIs.

System testing included load testing for 100,000 concurrent streams, cross-platform compatibility across 15 devices, and network resilience testing. **Acceptance testing** involved beta programme with 5,000 users providing feedback on user experience and content discovery.

The structured approach identified buffer time optimisations during system testing and UI improvements during acceptance testing, resulting in a 4.5-star launch rating.

QUICK QUIZ

Test Your Test Levels Knowledge

Question 1

Which test level is most appropriate for testing a payment gateway API integration?

- A) Unit Testing
- B) Integration Testing
- C) System Testing
- D) Acceptance Testing

Answer: B) Integration Testing

Question 2

In bottom-up integration testing, what are "drivers"?

- A) Lower-level modules
- B) Temporary code simulating higher modules
- C) Automation scripts
- D) Database connections

Answer: B) Temporary code simulating higher modules

Question 3

Which test level typically has the highest automation percentage?

- A) Acceptance Testing
- B) System Testing
- C) Integration Testing
- D) Unit Testing

Answer: D) Unit Testing (90%+)

Interview Keywords: Test Levels

Test Pyramid

Conceptual model suggesting more unit tests, fewer integration tests, and minimal UI tests for optimal coverage and speed.

UAT

User Acceptance Testing validates the system meets business needs and is ready for production deployment.

Stubs and Drivers

Temporary code used in integration testing to simulate unavailable modules during incremental testing.

Alpha/Beta Testing

Alpha: Internal testing by employees. Beta: External testing by selected real users before public release.

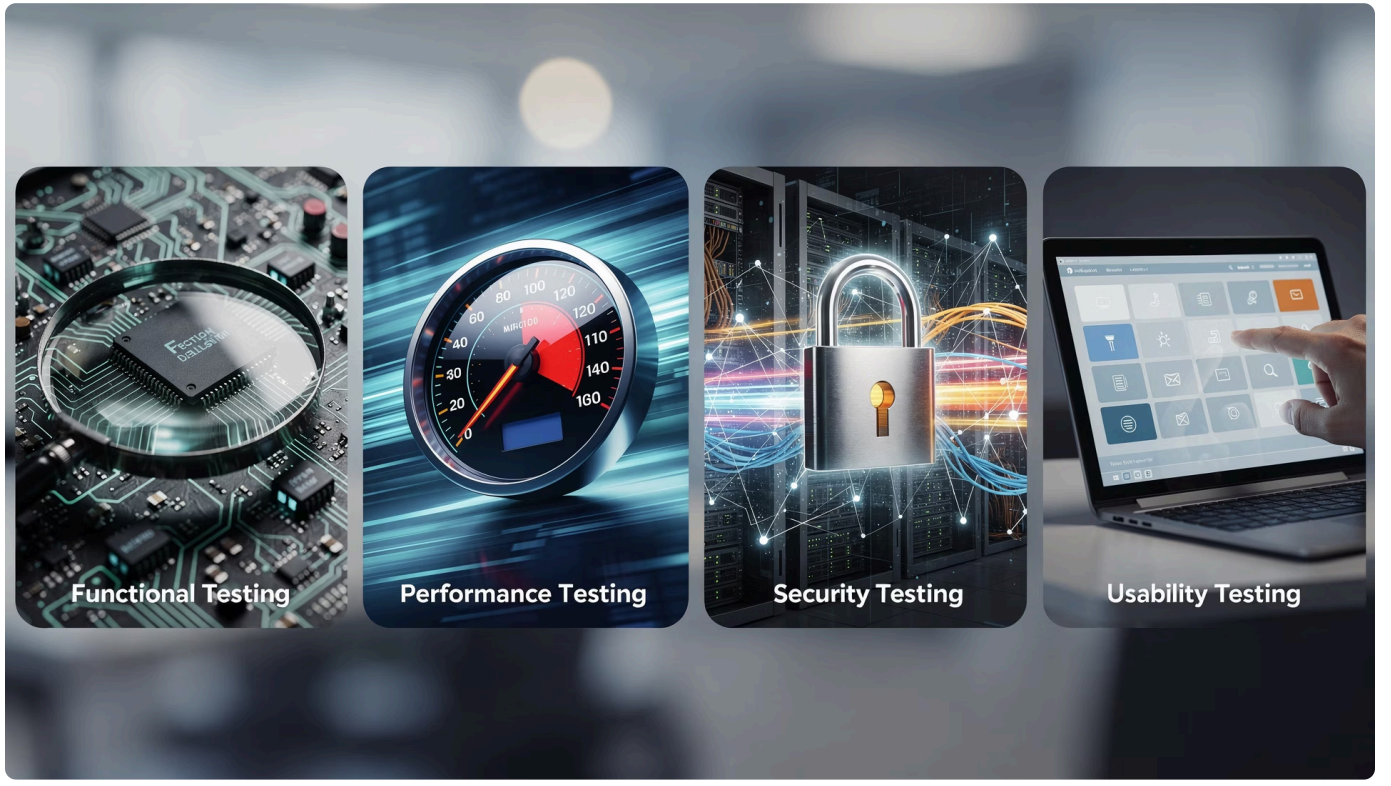
CHAPTER 3

Test Types

Test types define *what aspects* of the software are being tested, whilst test levels define *when* testing occurs. Test types can be applied at any test level.

The two primary categories are **functional testing** (verifying the system does what it should) and **non-functional testing** (verifying how well the system performs its functions).

Understanding various test types enables testers to design comprehensive test strategies that address all quality attributes: functionality, performance, security, usability, and more.



Functional Test Types

Smoke Testing

Quick verification that critical functions work. Determines if build is stable enough for detailed testing. Also called "build verification testing".

Sanity Testing

Narrow, focused testing after bug fixes or minor changes. Verifies specific functionality works as expected. Subset of regression testing.

Regression Testing

Ensures existing functionality remains intact after changes. Critical for continuous integration/deployment. Highly automatable test type.

Exploratory Testing

Simultaneous learning, test design, and execution. Tester explores application without predefined test cases. Uncovers unexpected issues.

Usability Testing

Evaluates user interface, navigation, and overall user experience. Identifies confusing workflows or design issues.

Accessibility Testing

Ensures application is usable by people with disabilities. Validates WCAG compliance, screen reader support.

Non-Functional Test Types

Performance Testing

Measures speed, responsiveness, and stability under workload. Includes load, stress, and endurance testing.

Security Testing

Identifies vulnerabilities and ensures data protection. Covers penetration testing, vulnerability scanning, authentication validation.

Compatibility Testing

Verifies application works across browsers, devices, operating systems, and network environments.

Reliability Testing

Evaluates system stability over time. Tests mean time between failures (MTBF) and recovery capabilities.

Smoke vs Sanity vs Regression Testing

Characteristic	Smoke Testing	Sanity Testing	Regression Testing
Purpose	Build stability check	Verify specific fix	Ensure no side effects
Scope	Broad but shallow	Narrow and deep	Broad and deep
When	New build received	After minor change	After any code change
Executed By	Developers or testers	Testers	Testers
Documentation	Not documented	Not documented	Documented test cases
Automation	Usually automated	Usually manual	Highly automated
Time	Minutes	Hours	Hours to days

DETAILED COMPARISON

Performance Testing Sub-Types

Understanding Performance Testing Variants

Each performance test type serves a specific purpose in validating system behaviour under different conditions.

Load testing simulates expected user volumes to ensure the application meets performance benchmarks. **Stress testing** deliberately overloads the system to identify maximum capacity and failure points.

Spike testing validates the system handles sudden traffic surges, crucial for e-commerce sales events. **Endurance testing** runs sustained loads to identify memory leaks or degradation over time. **Volume testing** verifies database performance with large data sets.

INDUSTRY EXAMPLE

Test Types in Practice: Travel Booking Platform

Online Travel Agency Quality Assurance

A major travel platform applied multiple test types for their summer release. **Functional testing** covered search, filtering, booking, and payment flows across 50 airlines and 200 hotels. **Regression testing** ran 1,500 automated test cases after each build to protect existing functionality.

Performance testing simulated 50,000 concurrent users booking flights, measuring response times under peak load. **Security testing** included penetration testing for payment processing and SQL injection vulnerability scanning. **Compatibility testing** verified functionality across Chrome, Safari, Firefox, and Edge browsers plus iOS and Android apps.

Usability testing with 30 participants identified confusing date picker interactions, leading to a redesign that increased conversion rates by 15%.

INDUSTRY EXAMPLE

Test Types: Social Media Platform

Comprehensive Testing Strategy

A social media company launching a new messaging feature implemented diverse test types.

Exploratory testing sessions uncovered edge cases where messages failed to send with special characters. **Accessibility testing** ensured screen reader compatibility and keyboard navigation for visually impaired users.

Localisation testing validated 25 language translations and cultural appropriateness. **Reliability testing** ran continuous 72-hour tests, discovering memory leaks in the notification service.

Key Results

23 critical defects found during exploratory testing that scripted tests missed. Accessibility improvements increased user base by 12%. Performance optimisations reduced server costs by 30%.

QUICK QUIZ

Test Your Test Types Knowledge

Question 1

Which test type would you use immediately after receiving a new build to check if testing can proceed?

- A) Regression Testing
- B) Smoke Testing
- C) Sanity Testing
- D) Exploratory Testing

Answer: B) Smoke Testing

Question 2

Testing an e-commerce site's behaviour during Black Friday sales would be classified as:

- A) Load Testing
- B) Stress Testing
- C) Spike Testing
- D) Volume Testing

Answer: C) Spike Testing

Question 3

Which testing approach involves simultaneous learning, test design, and execution?

- A) Regression Testing
- B) Sanity Testing
- C) Exploratory Testing
- D) Smoke Testing

Answer: C) Exploratory Testing

Interview Keywords: Test Types

Functional vs Non-Functional

Functional tests verify "what" the system does; non-functional tests verify "how well" it does it.

Positive vs Negative Testing

Positive tests verify system works with valid inputs; negative tests verify proper handling of invalid inputs.

Re-testing

Executing same test case on a defect fix to verify it's resolved. Different from regression testing.

Baseline

Performance metrics established during initial testing used for comparison in subsequent test cycles.

Bottleneck

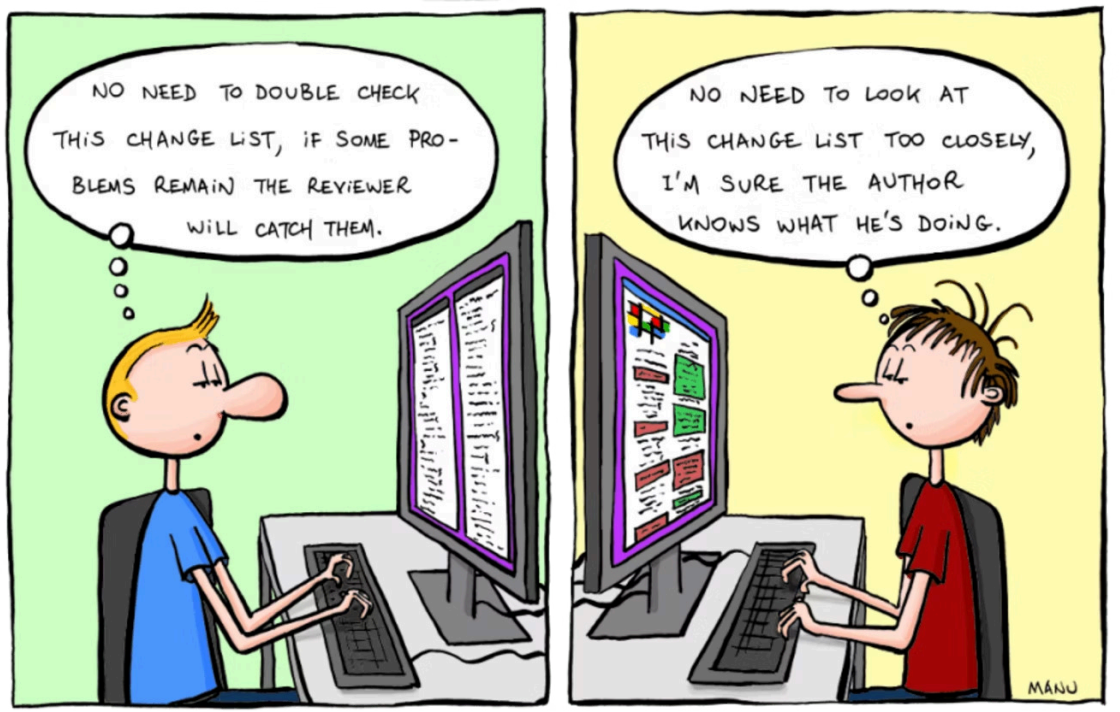
A component or resource that limits system performance under load conditions.

WCAG

Web Content Accessibility Guidelines - standards for making web content accessible to people with disabilities.

CHAPTER 4

Static vs Dynamic Testing



Testing can be broadly classified into two fundamental approaches based on whether code is executed.

Static testing examines code, documentation, or requirements without execution. It identifies defects early through reviews, inspections, and static analysis tools. Finds issues like coding standard violations, security vulnerabilities, and logical errors.

Dynamic testing involves executing code with test inputs and observing outputs. It validates actual runtime behaviour and is essential for verifying functional requirements, performance, and integration aspects.

Both approaches are complementary and necessary for comprehensive quality assurance.

Static Testing: Key Characteristics

No Code Execution

Testing performed by examining code, documents, or artefacts without running the application. Can start before code is complete.

Early Defect Detection

Identifies issues during requirements and design phases. Prevention costs significantly less than fixing defects in production.

Manual and Automated

Includes manual activities (reviews, walkthroughs) and automated tools (linters, static analysers, security scanners).

Preventative Nature

Aims to prevent defects from being introduced rather than finding them after coding. Improves overall code quality and maintainability.

Static Testing Techniques

Informal Reviews

Casual examination without formal process. Buddy checks, pair programming, quick desk checks for immediate feedback.

Walkthroughs

Author-led session explaining work product to team. Identifies defects through discussion, trains team members on new code.

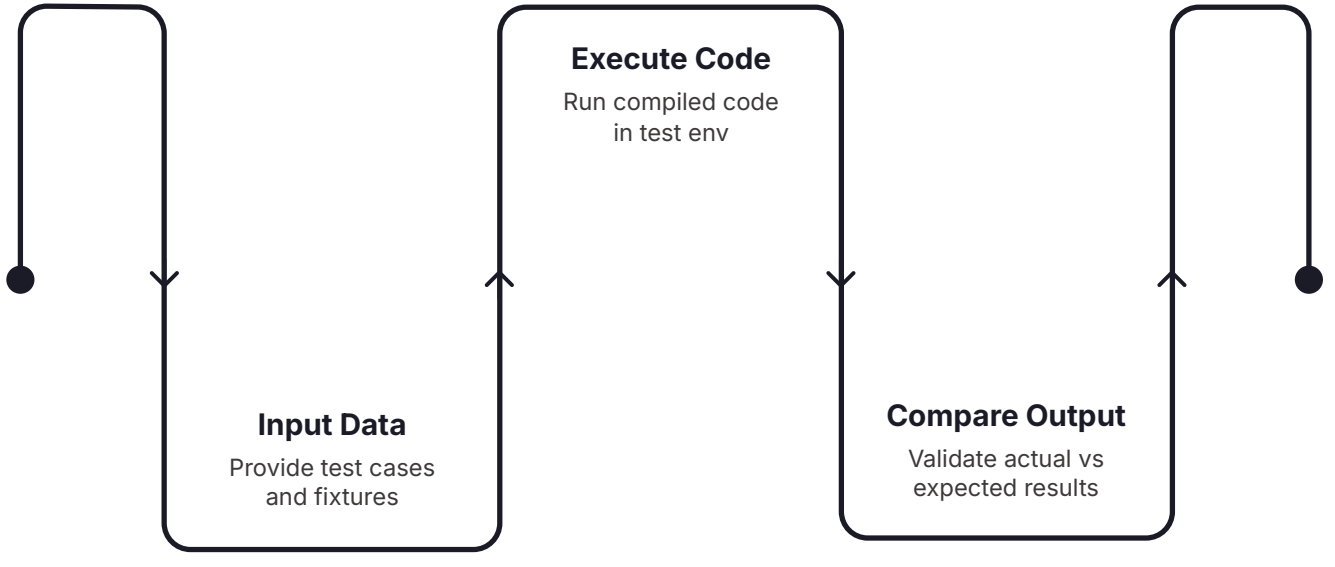
Inspections

Formal, rigorous process with defined roles. Uses checklists, measures defect density. Most effective but time-intensive method.

Static Analysis Tools

Automated code examination finding patterns, vulnerabilities, complexity issues. Tools like SonarQube, ESLint, FindBugs.

Dynamic Testing: Key Characteristics



Execution-Based Validation

Code must be compiled and executable. Tests are run in test environments that simulate production conditions.

Behaviour Verification

Validates the application does what it's supposed to do with actual data. Discovers runtime errors, integration issues, and performance problems.

Multiple Techniques

Encompasses black box, white box, and grey box testing approaches. Can be functional or non-functional.

Feedback Loop

Results inform developers about actual system behaviour. Failed tests lead to debugging and code fixes.

Static vs Dynamic Testing: Detailed Comparison

Aspect	Static Testing	Dynamic Testing
Execution	No code execution required	Code must be executed
When	Early in SDLC (requirements, design)	After code development
Defect Type	Logical errors, standards violations, ambiguity	Runtime errors, integration issues, performance
Cost	Lower (early detection)	Higher (late detection)
Coverage	100% code coverage possible	Cannot achieve 100% path coverage
Examples	Code reviews, inspections, walkthroughs	Unit, integration, system, UAT
Tools	SonarQube, ESLint, Checkstyle	Selenium, JUnit, JMeter
Approach	Prevention	Detection

INDUSTRY EXAMPLE

Static Testing Success: Financial Services

Investment Banking Application

A major bank implemented rigorous static testing for a trading platform handling millions of transactions.

Requirements reviews identified 45 ambiguities in specifications before coding began, saving an estimated 200 development hours. **Design inspections** caught architectural flaws that would have caused scalability issues.

Code reviews detected 127 potential security vulnerabilities using OWASP guidelines. **Static analysis tools** identified 89 code smells and 23 critical security issues automatically.

Impact

Defect density reduced by 60% compared to previous projects. 75% of defects found before dynamic testing phase. Development cycle shortened by 3 weeks. Zero critical post-production defects in first 6 months.

INDUSTRY EXAMPLE

Dynamic Testing Excellence: Gaming Platform

Multiplayer Game Launch Testing

A gaming studio launching a multiplayer online game relied on extensive dynamic testing. **Functional testing** covered gameplay mechanics, inventory systems, and matchmaking across 500+ test cases. **Integration testing** validated server-client communication and third-party authentication services.

Performance testing simulated 100,000 concurrent players identifying server bottlenecks. **Load balancing** was optimised after stress tests revealed database connection pool limitations. **Compatibility testing** across 25 device configurations uncovered graphics rendering issues on older hardware.

Beta testing with 10,000 players provided real-world validation. The comprehensive dynamic testing approach resulted in a smooth launch with 98% positive reviews and minimal server downtime.

QUICK QUIZ

Test Your Knowledge: Static vs Dynamic

Question 1

Which statement about static testing is correct?

- A) Requires executable code
- B) Can find runtime errors
- C) Can be performed on requirements documents
- D) Always requires automated tools

Answer: C) Can be performed on requirements documents

Question 2

Code reviews and inspections are examples of:

- A) Dynamic testing only
- B) Static testing only
- C) Both static and dynamic
- D) Neither

Answer: B) Static testing only

Question 3

Which defect type is typically found by dynamic testing?

- A) Coding standard violations
- B) Requirements ambiguity
- C) Memory leaks during execution
- D) Design inconsistencies

Answer: C) Memory leaks during execution

CHAPTER 5

Black Box vs White Box Testing

Testing techniques can be classified based on the tester's knowledge of internal code structure and implementation details.

Black box testing treats the application as a "black box" where internal workings are unknown. Testers focus on inputs and outputs without knowing implementation. This technique validates functional requirements and user perspectives.

White box testing requires knowledge of internal code, logic, and structure. Testers design cases based on code paths, branches, and conditions. This technique ensures thorough code coverage and logical correctness.

Both approaches are complementary, providing different perspectives on software quality.

Black Box Testing Techniques

1

Equivalence Partitioning

Divides input data into valid and invalid partitions. Test one value from each partition assuming all values behave similarly.

2

Boundary Value Analysis

Tests values at boundaries of equivalence partitions. Most defects occur at edges: minimum, maximum, just below, just above.

3

Decision Table Testing

Systematically tests combinations of conditions and their resulting actions. Useful for complex business logic with multiple conditions.

4

State Transition Testing

Models system behaviour as states and transitions. Tests valid and invalid state changes. Ideal for workflow-based applications.

5

Use Case Testing

Derives test cases from use cases representing user interactions. Ensures system meets real-world user scenarios.

6

Error Guessing

Experience-based technique where testers predict likely error areas. Leverages tester knowledge of common failure patterns.

White Box Testing Techniques

Statement Coverage

Ensures every statement in code executes at least once. Basic coverage metric but doesn't guarantee thorough testing. Formula: (Executed Statements / Total Statements) × 100

Branch Coverage

Tests all decision branches (true/false) in code. More thorough than statement coverage. Ensures both IF and ELSE paths execute. Formula: (Executed Branches / Total Branches) × 100

Condition Coverage

Tests all individual conditions in decision statements. Each boolean sub-expression evaluated to true and false independently. More rigorous than branch coverage.

Path Coverage

Executes all possible paths through code. Most thorough but often impractical for complex code with loops. Tests all combinations of branches.

Black Box vs White Box: Comprehensive Comparison

Aspect	Black Box Testing	White Box Testing
Knowledge	No internal code knowledge needed	Requires code and logic understanding
Performed By	Testers, end users	Developers, technical testers
Focus	Functionality and requirements	Code structure and logic
Test Basis	Requirements, specifications	Code, design documents
Coverage	Requirements coverage	Code coverage (statements, branches)
When	Higher test levels (system, UAT)	Lower test levels (unit, integration)
Advantages	User perspective, no bias from code	Thorough, finds hidden errors
Limitations	Cannot test unused code, limited paths	Time-consuming, requires coding skills

CASE STUDY

End-to-End Case Study: E-Learning Platform Launch

Project Overview

An education technology company developing a comprehensive e-learning platform for 50,000 students. The platform includes course management, video streaming, assessments, progress tracking, and payment processing. Timeline: 6 months from requirements to production.

STLC Application

Requirement Analysis (2 weeks): Identified 320 testable requirements, created RTM mapping requirements to test cases, assessed automation feasibility at 65%. **Test Planning (1 week):** Risk-based approach prioritising payment and assessment modules, allocated 8 testers, planned 4 test environments.

Test Case Development (3 weeks): Created 850 test cases (550 manual, 300 automated), prepared test data for 100 courses and 500 users. **Environment Setup (1 week):** Configured staging environment mirroring production, integrated with test payment gateway.

CASE STUDY

E-Learning Platform: Testing Strategy and Results

Test Levels and Types Applied

Unit Testing: Developers wrote 2,400+ unit tests achieving 87% code coverage. **Integration Testing:** Validated LMS-payment gateway, video CDN, and SSO authentication interfaces using bottom-up approach.

System Testing: End-to-end testing of complete user journeys from registration to course completion. **Performance Testing:** Load tested 10,000 concurrent video streams, stress tested assessment submission under peak load.

Security Testing: Penetration testing for payment module, vulnerability scanning identified and fixed 12 issues. **Usability Testing:** 25 students and 5 instructors validated interface and workflows.

Techniques and Results

Static Testing: Code reviews caught 89 issues before testing. Requirements reviews clarified 34 ambiguities early.

Black Box: Equivalence partitioning for input validation, boundary value analysis for grade calculations, state transition testing for course enrolment workflow.

White Box: Achieved 85% branch coverage for critical modules, path coverage for payment processing logic.

Outcome: Found 234 defects (18 critical, 67 major). Zero critical defects in production. 95% user satisfaction. Platform successfully scaled to 50,000 users within first year.