

# **SOFTWARE QUALITY ASSURANCE PROCESSES AND ACTIVITIES**

**Chapter 5**  
**Software Quality Assurance Processes and Activities**

# INTRODUCTION

Software Quality Assurance (SQA) is an integral part of software development, ensuring that products meet defined quality standards and user expectations.

This chapter provides an in-depth examination of SQA processes and activities, including

- verification and validation techniques
- the role of SQA in various Software Development Life Cycle (SDLC) models
- reviews and audits
- comprehensive testing strategies
- Understanding these elements ensures robust software engineering practices that contribute to reliability, maintainability, and performance optimization.

# SQA IN THE SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

SQA plays a critical role throughout the SDLC, helping to identify and mitigate issues at every stage.

- **Waterfall Model:** SQA is applied in sequential phases, with rigorous documentation and reviews at each stage.
- **Agile Model:** SQA integrates into iterative development cycles, using continuous testing, automated tools, and cross-functional team collaboration.
- **DevOps Model:** Focuses on continuous integration and continuous delivery (CI/CD), with automated testing and real-time monitoring ensuring rapid, high-quality releases.
- **V-Model:** Also known as the Verification and Validation model, it emphasizes parallel testing at each development stage.

# REVIEWS AND AUDITS

Reviews and audits are formal techniques used to evaluate software processes, documents, and products to maintain quality standards.

- **Code Reviews:**
  - Conducted by peers or dedicated review teams to identify coding errors and enforce best practices.
  - Types include pair programming, formal inspections, and lightweight reviews using version control tools.
- **Technical Reviews:**
  - Focus on assessing system architecture, design patterns, and implementation choices.
  - Helps ensure adherence to non-functional requirements such as scalability and maintainability.
- **Walkthroughs:**
  - Semi-formal process where a developer presents code or documentation to a group for constructive feedback.
- **Audits:**
  - Conducted internally or externally to verify compliance with industry standards such as ISO 9001, CMMI, or IEEE guidelines.

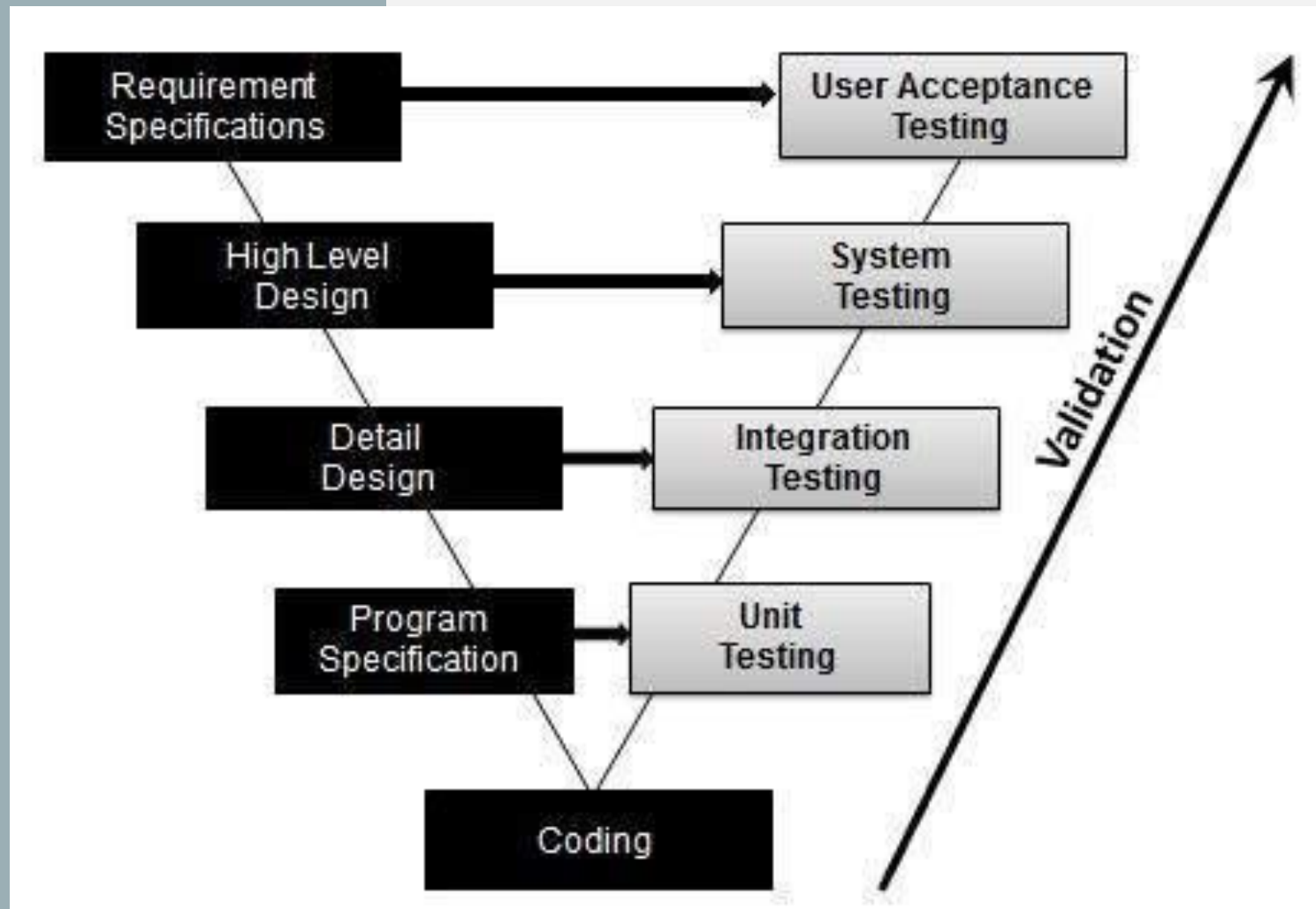
# TESTING STRATEGIES

Testing is a core activity in Software Quality Assurance (SQA), providing structured approaches to detect and fix defects, ensuring software reliability and compliance with requirements.

- **Unit Testing:** Focuses on individual components to ensure correctness in isolation.
- **Integration Testing:** Ensures that different modules communicate effectively.
- **System Testing:** Validates the entire software system against requirements.
- **Acceptance Testing:** Conducted by end-users to verify software meets business needs.
- **Regression Testing:** Ensures new changes do not introduce defects into existing functionality.
- **Automation in Testing:** Reduces manual effort and improves efficiency.

# VERIFICATION AND VALIDATION

Verification and validation are two fundamental components of the SQA process that ensure software correctness and usability.



# VERIFICATION

**Verification:** This is a static process that involves evaluating work products such as requirement documents, design specifications, and source code to confirm that they align with the predefined standards.

- It answers the question, "**Are we building the product right?**"
- **Key Verification Techniques:**
  - **Inspections:** A formal peer review process where documents or code are examined for errors, inconsistencies, and deviations from standards.
  - **Reviews:** A structured process involving multiple stakeholders to examine work artifacts.
  - **Walkthroughs:** Informal sessions where developers or designers present their work for feedback.
  - **Prototyping:** Creating early models of software to validate assumptions and refine requirements.

# VERIFICATION (STATIC TESTING)

## **What is Static Testing?**

- Static Testing also known as Verification testing or Non-execution testing is a type of Software Testing method that is performed in the early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily. The errors that cannot be found using Dynamic Testing, can be easily found by Static Testing.
- Static can be done manually or with the help of tools to find bugs and improve the quality of the software.
- It helps to find errors in the early stage of development which is also called the verification process.
- It enhances maintainability and ultimately saves time and money in the long run.



# NEED FOR STATIC TESTING

Static testing is needed whenever the following situations are encountered while testing an application or software:

- **Increased software size:** Static testing is required to get free from bugs in the early stages of development as with testing activity the size of the software increases which is difficult to handle due to a reduction in the productivity of the code coverage.
- **Dynamic testing is expensive:** Dynamic testing is more expensive than static testing as dynamic testing uses test cases that have been created in the initial stages and there is also a need to preserve the implementation and validation of the test cases which takes a lot of time from test engineers
- **Dynamic testing is time-consuming:** Static testing is required as dynamic testing is a time-consuming process.
- **Bugs detection at early stages:** Static testing is helpful as it finds bugs at early stages, while dynamic testing finds bugs at later stages which makes it time-consuming and costly to fix the bugs.
- **Improvement of development productivity:** Static testing helps to identify bugs early in the software development thus it helps to reduce the flaws during production and increase development productivity.

# FEATURES TESTED IN STATIC TESTING

## Features Tested in Static Testing

- **Unit Test Cases:** It ensures test cases are complete, written in the correct manner, and follow the specified standards.
- **Business Requirements Document:** It verifies that all business requirements are clearly mentioned in the documentation.
- **Use Cases:** It examines the use cases so that they accurately represent user interactions with the system.
- **Prototype:** It reviews the prototype to make sure that it accurately represents the main design and functionality.
- **System Requirements:** It check the system requirements document for complete accuracy.
- **Test Data:** It reviews the test data to ensure it is complete and covers all possible input scenarios.
- **Traceability Matrix Document:** This ensures that all requirements are mapped to corresponding test cases.
- **Training Guides:** It reviews training materials to make sure that they accurately reflect the system functionality and user procedures.
- **Performance Test Scripts:** It examines performance test scripts to ensure they cover all critical performance aspects.

# VALIDATION

**Validation:** This is a dynamic process that involves executing the software to ensure it meets business and user requirements. It answers the question, "Are we building the right product?"



- **Key Validation Techniques:**
  - **Functional Testing:** Ensures that the software meets functional requirements.
  - **Usability Testing:** Evaluates the user experience and ease of use.
  - **Performance Testing:** Assesses the responsiveness, stability, and scalability of the software.
- **Security Testing:** Identifies vulnerabilities to protect against potential threats.

# WHITE BOX AND BLACK BOX TESTING

White-box and black-box testing are two fundamental **software testing techniques** used to ensure software quality. They differ in **methodology, scope, and knowledge requirements** and are classified based on different criteria.

- **White-Box Testing** is a **structural testing technique** where the tester examines the **internal code, logic, and structure** of the software to ensure correctness. It is also called **Clear-Box, Glass-Box, or Open-Box Testing**.
- **Key Characteristics:**
  - **Code-level access:** Testers need access to the source code and must understand programming logic.
  - **Focus on logic and flow:** It validates control structures such as branches, loops, and conditions.
  - **Code coverage measurement:** Various metrics like statement coverage, branch coverage, and path coverage are used to quantify how thoroughly the code is tested.
  - **Done by developers or technically skilled testers:** Often used in **unit testing** and **integration testing** phases.

# WHITE BOX COMMON TECHNIQUES

- **Statement Coverage –**
  - Ensures that **each line of code** is executed at least once. It's the most basic form of code coverage.
  - **Objective:**  
To guarantee that **no part of the code remains untested**.
  - Example : **Test Case for Statement Coverage:**
  - input: 5 \_Execution Path: num > 0 is true → return True →  Statement covered
  - Input -3 \_Execution Path: num > 0 is False → return False →  Statement covered

# WHITE BOX COMMON TECHNIQUES

- **Branch Coverage**

- Tests **all possible branches** in the control structures (like if, while, for). Each decision point should result in both true and false at least once.
- Objective: To ensure that every **decision-making construct** is tested under all possible outcomes
- Example :

- **Required Test Cases:**

- Input: 2 → True branch
- Input: 3 → False branch

```
def is_even(n):  
    if n % 2 == 0:  
        return True  
    else:  
        return False
```

# WHITE BOX COMMON TECHNIQUES

- **Path Coverage**

- Ensures that **every possible route** through a given section of code is executed.
- **Objective:** To verify **all independent paths**, especially when multiple conditions are nested.
- Example
- **Paths to be tested :**
  - score = 95 → “Excellent”
  - score = 80 → “Good”
  - score = 60 → “Needs Improvement”

```
def categorize_score(score):  
    if score >= 90:  
        print("Excellent")  
    elif score >= 75:  
        print("Good")  
    else:  
        print("Needs  
Improvement")  
    print("Evaluation Done")
```

# WHITE BOX COMMON TECHNIQUES

- **Mutation Testing**

- Involves creating **mutants** (modified versions of the source code) by introducing small changes like replacing operators. Then, the test suite is run to see if it can **detect the error**.
- **Objective:** To **measure the effectiveness** of test cases — good test cases should "kill" mutants by failing when encountering faulty code.
- Example

Original Code:  
`def multiply(a, b):  
 return a * b`

Mutant Code:  
`def Multiply(a, b):  
 return a + b`

Test Case:  
`assert multiply(2, 3) == 6 # This will fail for the mutant`

- **Result:**  
**Mutation detected** → test suite is effective.  
**Mutation not detected** → test suite needs improvement.



# BLACK BOX TESTING

**Black-box testing** is a software testing technique that focuses on **validating the external functionality** of a system. The tester is not concerned with the internal workings or source code but only with how the software behaves with given **inputs** and **expected outputs**.

## Key Characteristics:

Code Access: No access to or knowledge of source code required

Focus: Inputs, expected outputs, user interaction, and system behavior

Type of Testing: Mostly functional, but can also be used for non-functional tests

Who Performs It?: Testers, QA engineers, user acceptance testers

Main Testing Levels: System testing, acceptance testing, integration testing

Goal: Validate that the system does what it's supposed to do

White-box testing ensures **internal code correctness**, while black-box testing verifies **software functionality** from a user's perspective. Both are **essential** in Software Quality Assurance (SQA) and are often **combined (Gray-Box Testing) for comprehensive validation**.

# BLACK BOX TESTING

**Equivalence Partitioning (EP)** Inputs are divided into equivalence classes (or partitions) where the system is expected to behave similarly. Instead of testing every input, you test one value per class, assuming the rest behave the same.

- Use Case:  
Used when the input domain is large or infinite (e.g., form fields, numeric ranges, login credentials).

Examples:

1. Input: Age (valid range: 18–60)
  - Valid: 30
  - Invalid: 15, 65
2. Input: Email address
  - Valid: test@example.com
  - Invalid: test.com, @example.com
3. Input: Payment amount (100–10000 EGP)
  - Valid: 1000
  - Invalid: 50, 15000

# BLACK BOX TESTING

**Boundary Value Analysis (BVA)** Focuses on testing the edges of input domains. Errors often occur at boundaries.

- Use Case: When inputs have minimum and maximum limits.
- Examples:
  1. Input: Age (18–60)
    - Test: 17, 18, 19 and 59, 60, 61
  2. Password length (8–20 characters)
    - Test: 7, 8, 9 and 19, 20, 21
  3. Online order quantity (1–5 items)
    - Test: 0, 1, 2 and 4, 5, 6

# BLACK BOX TESTING

**Decision Table Testing is** Used when software behavior depends on combinations of multiple conditions.

- Use Case: Ideal for business rules, access control, or discount logic.
- Examples:
  1. Login Access
    - Valid username/password: Access granted
    - Invalid combinations: Access denied
  2. Discount Logic
    - Member + Coupon + >1000 EGP: Discount applied
    - No conditions met: No discount
  3. Banking Security
    - Biometric + OTP: Access granted
    - Others: Blocked

# BLACK BOX TESTING

- **State Transition Testing** Tests systems where different inputs cause state changes.
- Use Case: Workflow-based systems (ATMs, elevators, forms).
- Examples:
  1. ATM Card
    - Card Inserted + Correct PIN: Main Menu
    - Wrong PIN 3x: Card Blocked
  2. Order Status
    - Placed → Paid → Shipped → Delivered
  3. User Account
    - Active → Request Deletion → Pending → Deleted or Active