Unit V:Software Testing Fundamentals

Software testing is the fundamental process of verifying and validating a software application to ensure it meets technical and business requirements and is free of defects. It is a critical part of the Software Development Life Cycle (SDLC) that reduces risk, improves product quality, and enhances customer satisfaction.

Importance of software testing

- **Early defect detection**: Finding and fixing bugs in the early stages of development is significantly cheaper and easier than addressing them after the software has been deployed.
- **Improved product quality**: Testing ensures that the software meets quality standards, making it more reliable, efficient, and usable.
- **Customer satisfaction**: A reliable and high-performing application that fulfills user expectations can lead to greater customer satisfaction and brand loyalty.
- **Security**: systematic testing uncovers security vulnerabilities and loopholes, protecting user data and preventing malicious attacks.
- **Cost-effectiveness**: While testing requires resources, the cost of fixing post-release failures is often much higher than the investment in early testing.

Software Testing Life Cycle (STLC)

The STLC is a structured sequence of activities that ensures a systematic approach to testing. The typical phases include:

1. **Requirement Analysis**: The testing team analyzes the software requirements to identify testable features, prioritize testing efforts, and prepare a Requirement Traceability Matrix (RTM).

2. **Test Planning**: Test managers create a test plan document that defines the testing scope, strategy, tools, resources, and timelines.

3. **Test Case Development**: Testers write detailed test cases, prepare test data, and review them for accuracy and traceability to requirements.

4. **Test Environment Setup**: The hardware, software, and network are configured to take off the production environment. A smoke test is often run to verify the setup.

5. **Test Execution**: Test cases are executed, and results are logged. Any deviations from expected behavior are reported as defects to the development team.

6. **Test Cycle Closure**: The testing team analyzes the test results, prepares a test summary report, and documents lessons learned for future projects.

Core testing types with examples

Software testing can be categorized by the underlying methodology and by what is being tested.

Based on test approach

- **Manual Testing**: Performed by human testers who manually execute test cases without automation tools. It is flexible and good for exploratory testing.

- **Example**: A tester manually enters a username and password into a login form to verify that they can access their account.

- **Automated Testing**: Uses scripts and tools to execute repetitive test cases quickly and efficiently, especially useful for large systems and regression testing.
    - **Example**: A script is created using a tool like Selenium to automatically test that all the links on a website's navigation bar are functional.

White box texting

White box testing is a software testing method that focuses on the internal structure, logic, and code of a software application. Unlike black box testing, which treats the software as a sealed system, white box testing requires knowledge of the inner workings to design test cases and scrutinize the internal operations.

It is also known by other names, including structural, clear box, glass box, or transparent box testing.

Fundamentals of white box testing

- **Access to source code:** Testers have complete access to the internal code, design documents, and architecture of the application. This allows them to create tests based on the actual implementation rather than just the stated requirements.
- **Code coverage analysis:** A primary goal is to ensure that a high percentage of the code is executed by the test cases. Techniques are used to measure and maximize code coverage, including statement, branch, and path coverage.
- **Verification of internal logic:** The main focus is on verifying that the program's logic and control flow are correct. This includes checking conditional statements (e.g., if-else), loops, and internal functions.

- **Early bug detection:** White box testing is often performed early in the development cycle, typically by developers during unit testing. This allows for the detection and correction of bugs at a stage when they are less expensive and easier to fix.
- **Requires programming skills:** Testers must possess programming knowledge to understand the code, identify testable paths, and write the necessary test cases.

Key techniques and examples

1. Statement coverage

This technique ensures that every executable statement or line of code is tested at least once. It verifies that no line of code is "dead" or unused.

Example
Consider the following pseudo-code for a function that checks if a user is eligible for a senior discount:

```
function check_senior_discount(age, has_loyalty_card):
  discount_eligible = false
  if age >= 65:
    discount_eligible = true
  if has_loyalty_card:
    discount_eligible = true
  return discount_eligible
```

To achieve 100% statement coverage, you need two test cases:

- **Test Case 1:** An input that executes the first if statement (e.g., age = 70, has_loyalty_card = false).
- **Test Case 2:** An input that executes the second if statement (e.g., age = 45, has_loyalty_card = true).

2. Branch coverage (or decision coverage)

This technique ensures that every branch (true and false outcomes) of each decision point in the code is executed at least once.

Example

Using the same check_senior_discount function:

```
function check_senior_discount(age, has_loyalty_card):
  discount_eligible = false
  if age >= 65:
    discount_eligible = true
  if has_loyalty_card:
    discount_eligible = true
  return discount_eligible
```

To achieve branch coverage, you would need four test cases to cover all combinations of the two conditions:

- **Test Case 1:** age = 70, has_loyalty_card = true (both conditions are true).
- **Test Case 2:** age = 70, has_loyalty_card = false (first condition true, second false).
- **Test Case 3:** age = 45, has_loyalty_card = true (first condition false, second true).
- **Test Case 4:** age = 45, has_loyalty_card = false (both conditions are false).

3. Loop testing

This technique focuses specifically on validating the correct behavior of loops within the code. Test cases are designed to test the loop for zero, one, and multiple iterations, as well as for boundary conditions.

Example

Consider a function that calculates the sum of a list of numbers:

```
function calculate_sum(numbers):
  sum = 0
  for number in numbers:
    sum += number
  return sum
```

Test cases for loop testing would include:

- **Zero iterations:** Pass an empty list []. The expected output is 0.
- **One iteration:** Pass a list with one item [5]. The expected output is 5.
- **Multiple iterations:** Pass a list with several items [1, 2, 3]. The expected output is 6.

Basis Path Testing

Basis path testing is a white-box testing technique in software engineering that aims to achieve thorough code coverage by testing a set of linearly independent paths through a program's control flow. It focuses on the internal structure of the code, making it a valuable tool

for developers at the unit level.

Fundamentals of Basis Path Testing:

Control Flow Graph (CFG) Construction:

The first step involves creating a graphical representation of the program's control flow. This graph consists of nodes (representing blocks of code) and edges (representing the flow of control between these blocks). Decision points (e.g., if statements, while loops) lead to multiple outgoing edges.

Cyclomatic Complexity Calculation:

This metric, often attributed to McCabe, quantifies the logical complexity of a program and determines the minimum number of independent paths that must be tested. It can be calculated using various formulas, such as V(G) = E - N + 2P (where E is the number of edges, N is the number of nodes, and P is the number of connected components) or by counting regions in the CFG.

Identification of Independent Paths:

Based on the cyclomatic complexity, a basis set of independent paths is identified. Each path in this set introduces at least one new edge or decision that has not been covered by previous paths.

Test Case Design:

Test cases are then designed to execute each of these independent paths. This involves setting up specific input conditions and data that will force the program to traverse the intended path through the CFG.

Example:
Consider the following simple code snippet:
Python

```
def calculate_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    else:
        return "C"
```

Steps for Basis Path Testing:

- **Control Flow Graph:**
    - Node 1: def calculate_grade(score):
    - Node 2: if score >= 90:
    - Node 3: return "A"
    - Node 4: elif score >= 80:
    - Node 5: return "B"

- Node 6: else:
- Node 7: return "C"

Edges connect these nodes based on the control flow. For example, from Node 2, there are edges to Node 3 (if true) and Node 4 (if false).

Cyclomatic Complexity:

Using the formula V(G) = E - N + 2P, or by counting regions, the cyclomatic complexity for this example would be 3 (representing the three distinct grade paths).

Independent Paths:

- Path 1: Node 1 -> Node 2 (true) -> Node 3 (Score >= 90)
- Path 2: Node 1 -> Node 2 (false) -> Node 4 (true) -> Node 5 (80 <= Score < 90)
- Path 3: Node 1 -> Node 2 (false) -> Node 4 (false) -> Node 6 -> Node 7 (Score < 80)

Test Cases:

- Test Case 1: calculate_grade(95) (Expected: "A")
- Test Case 2: calculate_grade(85) (Expected: "B")
- Test Case 3: calculate_grade(75) (Expected: "C")

By designing test cases for each independent path, basis path testing ensures that all logical conditions and branches within the code are exercised, leading to improved code coverage and the detection of potential errors.

Control structure testing

Control structure testing is a white-box testing technique that validates a program's internal logic and flow by testing its decision statements, loops, and

data flows. It is a fundamental part of structural testing and requires knowledge of the source code and its internal implementation.

Key Techniques and Examples

The main techniques used in control structure testing include Condition Testing, Data Flow Testing, and Loop Testing.

1. Condition Testing

Condition testing focuses on ensuring that the logical conditions and decision-making statements (like if, else, switch) in a program are free from errors. Test cases are designed to make each condition evaluate to both true and false outcomes.

Example:

Consider the following pseudo-code for a discount eligibility check:

```
IF (age > 60 AND is_member == true) THEN
   apply_discount = true
ELSE
   apply_discount = false
END IF
```

To test this using condition testing (specifically, multiple-condition coverage), test cases should cover all combinations of the simple conditions age > 60 and is_member == true:

- **Test Case 1:** age = 65, is_member = true (Both true, apply_discount = true)
- **Test Case 2:** age = 65, is_member = false (One true, one false, apply_discount = false)

- **Test Case 3:** age = 55, is_member = true (One false, one true, apply_discount = false)
- **Test Case 4:** age = 55, is_member = false (Both false, apply_discount = false)

2. Loop Testing

Loop testing is a white-box technique that specifically targets the validity of loop constructs (for, while, do-while). Errors often occur at loop boundaries, so test cases focus on the number of iterations.

Example:

Consider a simple for loop that iterates n times, where n is the maximum allowable number of passes.

```
FOR i from 1 to n:
   perform_action()
END FOR
```

Test cases for this loop should include:

- **Skip the loop entirely** (e.g., set n=0 or make the condition false initially).
- **Traverse the loop only once** (e.g., set n=1).
- **Traverse the loop two times** (e.g., set n=2).
- **Traverse the loop m times** where m < n (e.g., set n=10, m=5).

- **Traverse the loop n-1, n, and n+1 times** (e.g., test the boundaries).

3. Data Flow Testing

Data flow testing selects test paths based on the locations where variables are defined and used (definition-use chains) within the program. The goal is to ensure variables are handled correctly and to uncover anomalies like uninitialized variables or defined variables that are never used.

Example:
Consider a code segment where variable X is defined in statement S and used later in statement S'. A test case would be designed to follow a path from S to S' without any redefinition of X in between to ensure the value flows correctly. This helps to identify issues such as if the calculation for X was skipped under certain conditions.

Black box testing

Black box testing is a software testing method in which the tester evaluates an application's external functionality and behavior without any knowledge of its internal code structure, implementation details, or design. The tester interacts with the software through its user interface or APIs, provides inputs, and verifies that the outputs match the expected results based on the software's requirements and specifications.

Key Concepts

- **Focus on functionality:** The primary goal is to ensure the software performs its intended tasks correctly from the end-user's point of view.
- **No internal knowledge required:** Testers do not need programming skills or access to the source code, which allows for an objective and unbiased assessment.

- **Requirements-based:** Test cases are created based on the software's functional and non-functional requirements documents.

Example

Consider testing the **login page** of a banking website.

1. **Identify Functionality:** The main function is to allow users to log in with valid credentials and reject invalid ones.

2. **Create Test Cases:**

Valid Input: Enter a correct username and a correct password.

- *Expected Result:* The user is successfully logged in and redirected to their account dashboard.

Invalid Username: Enter an incorrect username and a correct password.

- *Expected Result:* An error message is displayed: "Invalid username or password".

Invalid Password: Enter a correct username and an incorrect password.

- *Expected Result:* An error message is displayed, and the number of failed attempts is tracked (e.g., after a certain number of attempts, the account is locked).

1. **Boundary Conditions:** Leave the username and/or password fields blank.

- *Expected Result:* An error message prompting the user to enter credentials appears.

Execute and Verify: The tester runs these scenarios and compares the actual outcome with the expected outcome. If they match, the test case passes. If not, a bug is reported to the development team.

Common Techniques

Testers use specific techniques to reduce the number of test cases while ensuring good coverage:

- **Equivalence Partitioning:** Dividing input data into distinct "classes" or "partitions" where all values within a class are expected to produce the same result. Only one value from each class is tested.
- **Boundary Value Analysis (BVA):** Testing values at the edges of valid and invalid input ranges (e.g., for an age input field accepting values 18-30, testers would test 17, 18, 30, and 31).
- **Decision Table Testing:** Used for systems with complex business rules where the output depends on multiple combinations of inputs.
- **State Transition Testing:** Focusing on how the system changes from one state to another in response to specific events or inputs (e.g., an account locking after multiple failed login attempts).
- **Error Guessing:** Using the tester's experience and intuition to anticipate where defects might occur and designing test cases accordingly.

Size Oriented Metrics

Size-oriented metrics are derived by normalizing quality and productivity Point Metrics measures by considering the size of the software that has been produced. The organization builds a simple record of size measure for the software projects. It is built on past experiences of organizations. It is a direct

measure of software. This metric **measure** is one of the simplest and earliest metrics that is used for computer programs to measure size.

Size Oriented Metrics are also used for measuring and comparing the productivity of programmers. It is a direct measure of a Software. The size measurement is based on lines of code computation. The lines of code are defined as one line of text in a source file. While counting lines of code, the simplest standard is:

- Don't count blank lines
- Don't count comments
- Count everything else
- The size-oriented measure is not a universally accepted method.

A simple set of size measures that can be developed is given below:

Size = Kilo Lines of Code (KLOC)
Effort = Person / month
Productivity = KLOC / person-month
Quality = Number of faults / KLOC
Cost = $ / KLOC
Documentation = Pages of documentation / KLOC

Advantages of Size-Oriented Metrics

- **Easy to Use:** Simple to apply by counting things like lines of code or function points, making it quick to estimate effort and costs.
- **Early Estimation:** Helps predict the size of a project during the planning phase, allowing for realistic budgets and timelines before coding starts.

- **Easy Comparison Across Projects:** Standardized metrics make it easy to compare projects, aiding in benchmarking and cost prediction.
- **Builds Historical Data:** Tracks past project data (like cost per line of code), improving the accuracy of future estimates.
- **Reduces Guesswork:** Uses concrete measurements instead of guesses, leading to more consistent and reliable results.
- **Great for Large Projects:** Breaks large projects into manageable chunks, simplifying progress tracking and cost management.

Example of Size-Oriented Metrics

For a size oriented metrics, software organization maintains records in tabular form. The typical table entries are: Project Name, LOC, Efforts, Pages of documents, Errors, Defects, Total number of people working on it.

Project Name	LOC	Effort	Cost ($)	Doc. (pages)	Errors	Defects	People
ABC	10, 000	20	170	400	100	12	4
PQR	20, 000	60	300	1000	129	32	6
XYZ	20, 000	65	522	1290	280	87	7

Function-oriented metrics

Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value, focusing on the software's behavior and user-visible services rather than implementation details like lines of code. The most widely used example is **Function Point (FP) analysis**.

Core Concepts

- **User-centric:** These metrics measure software characteristics from the user's perspective, based on what the user requests and receives.
- **Language-independent:** They are not tied to a specific programming language, making them useful for comparing projects using different technologies.
- **Early Estimation:** Since they rely on functional requirements, they can be calculated earlier in the project lifecycle than code-based metrics, aiding in initial effort and cost estimations.

Measurement Parameter	Definition
External Inputs (EI)	Each unique user input type that provides distinct application-oriented data to the software (e.g., a data entry screen or form).
External Outputs (EO)	Each unique output type that provides application-oriented information to the user (e.g., reports, error messages, or screens).
External Inquiries (EQ)	Each unique on-line input that results in the generation of an immediate on-line output/response

	(e.g., a search query).
Internal Logical Files (ILF)	A logical grouping of data or a master file maintained within the application's boundary (e.g., a database table).
External Interface Files (EIF)	A logical grouping of data that is referenced by the application but maintained by another external application (e.g., an interface file from another system).

Primary Example: Function Points (FP) Analysis

The Function Point (FP) metric, developed by Albrecht at IBM, is the primary function-oriented metric. It quantifies the software's size by counting and weighting five key information domain characteristics:

Calculation Steps (Simplified)

1. **Count** the occurrences of each parameter (EI, EO, EQ, ILF, EIF).
2. **Assign complexity** (simple, average, or complex) to each count based on established criteria.
3. **Apply weighting factors** to these counts to get an initial function point total.
4. **Adjust for overall system complexity** using 14 global adjustment factors (e.g., data communications, performance criticality, reusability) to arrive at the final Function Point count.

Other Examples of Function-Oriented Metrics

- **Feature Points:** An extension of function points, often used for systems with significant internal processing rather than user interaction, such as operating systems or real-time software.
- **Use Case Points (UCP):** Used in agile methodologies, UCPs measure the system's size based on the number and complexity of use cases and actors involved.
- **Object Points (OP):** Measures size based on the number and complexity of objects, attributes, and methods defined in object-oriented design.

Metrics for Software quality

Software quality metrics include **defect density** (defects per lines of code), **test coverage** (percentage of code covered by tests), and **Mean Time to Recovery (MTTR)** (average time to restore service after a failure). Other key metrics are **lead time for changes** (time from commit to deployment), **change failure rate** (percentage of deployments causing failures), and **customer-reported bugs** (number of unique defects reported by users).

Product quality metrics

- **Defect Density:** Measures the number of defects per unit of size (e.g., lines of code or function points).
    - **Example:** 10 defects in 20,000 lines of code results in a defect density of 0.5 defects per 1,000 lines of code.

- **Mean Time to Failure (MTTF)/Mean Time Between Failures:** The average time between one failure and the next.
    - **Example:** Used for safety-critical systems like avionics; an MTTF of 500 hours means the system is expected to fail, on average, every 500 hours of use.
- **Customer-Reported Bugs:** The number of unique defects reported by customers, which can be further analyzed by severity.
    - **Example:** 50 bugs were reported by users in the last quarter.
- **Defect Leakage:** The number of defects that escape the development team and are found by the customer or during user acceptance testing (UAT).
    - **Example:** 5 critical bugs were found in production that should have been caught during internal testing.

Process and performance metrics

- **Test Coverage:** The percentage of your codebase that is executed by your automated tests.
    - **Example:** A test coverage of 85% means that 85% of the application's code is being tested.
- **Lead Time for Changes:** The time it takes from a code commit to that code successfully running in production.
    - **Example:** It takes an average of 2 hours for a code change to go from commit to deployment.

- **Mean Time to Recovery (MTTR):** The average time it takes to restore service after a system failure.
    - **Example:** A server fails, and it takes, on average, 15 minutes to bring it back online.
- **Change Failure Rate:** The percentage of deployments that result in a failure and require remediation (e.g., a rollback or hotfix).
    - **Example:** Out of 100 deployments, 5 resulted in a failure, giving a 5% change failure rate.
- **Code Churn:** The rate at which code is being modified, added, or deleted over a specific period.
    - **Example:** A specific module of code has a high churn rate, indicating it is unstable and potentially error-prone.

Empirical Estimation Models

Empirical estimation models are mathematical formulas used in software engineering to predict the effort, cost, and duration of a project, based on historical data and observed metrics. These models are more reliable than guesswork, providing a data-driven foundation for project planning and risk management.

Core concepts of empirical estimation

- **Historical data:** The models are built on data collected from past projects, including metrics such as project size, team experience, development tools, and software complexity.
- **Statistical analysis:** Statistical techniques like regression analysis are used to analyze the historical data. This helps determine the relationship between project variables and outcomes, allowing for the derivation of predictive formulas.

- **Effort and size relationship:** The core principle is that a relationship exists between the size of a software product and the effort required to develop it. This is typically an exponential relationship, where effort increases at an accelerating rate as project size grows.
- **Customization:** No single model fits every project or organization. The constants within the formulas must be calibrated and adjusted to reflect a specific organization's environment, processes, and historical data.
- **Project drivers:** Sophisticated models incorporate "cost drivers" or "effort multipliers," which are characteristics of the project that can influence the final effort. These factors include product reliability, database size, and programmer capability.

Examples of empirical estimation models

1. Constructive Cost Model (COCOMO)

Developed by Barry Boehm, COCOMO is one of the most widely known empirical estimation models. It has evolved into several versions, including COCOMO II, to accommodate modern software practices.

The formula: The basic COCOMO formula for estimating effort (in person-months) is:

$Effort = a \times (Size)b$

$Size$ is measured in thousands of lines of code (KLOC).

a and b are constants determined by the project's development mode

Development modes: The original COCOMO model classifies projects into three modes, each with different values for a and b:

- **Organic:** Small, simple projects developed by small, experienced teams with flexible requirements.
- **Semi-detached:** Intermediate projects with a mix of team experience and requirements.

- **Embedded:** Complex, large-scale projects with tight hardware, software, and operational constraints.

Example of Basic COCOMO:
A semi-detached project is estimated to have a size of 50,000 lines of code (50 KLOC). For this mode, the constants area a=3.0 and b=1.12.

$Effort$=3.0×(50)1.12≈202 person-months.

- This suggests the project would require approximately 202 person-months of effort.

2. The Software Equation

This is a dynamic, multivariable model that uses a formula derived from data collected on thousands of software projects. It assumes a specific distribution of effort over a project's lifespan.

The formula:

$$E = B \times (Size/P)^{t^3/333}$$

- E = Effort (in person-months or person-years)
- t = Project duration (in months or years)
- B = A special skills factor (0.16 to 0.39)
- P = A productivity parameter reflecting development practices (with typical values ranging from 2,000 for embedded software to 28,000 for business applications).

Example of the Software Equation:
A business application with an estimated size of 100 KLOC is being developed.

- Assume the duration (t) is 12 months, the skills factor (B) is 0.20, and the productivity parameter (P) is 28,000.
- $Effort = 0.20 \times (100000/28000)^{12^3/333} \approx 13$ person-months.
- This model can also be rearranged to estimate project duration based on effort, or size

Software Quality Management

Software Quality Management (SQM) is a systematic process of ensuring that software products meet or exceed customer expectations and regulatory requirements. It is integrated throughout the entire Software Development Life Cycle (SDLC) to build quality into the product from the start rather than just testing for it at the end.

Core SQM concepts

SQM is composed of three interconnected techniques: quality assurance, quality planning, and quality control.

1. Quality Assurance (QA)

This is a proactive, process-oriented approach focused on preventing defects. QA ensures the development processes themselves are sound, so that the likelihood of producing a low-quality product is minimized.

- **Concept:** Establishing and implementing reliable processes, standards, and methodologies that the entire team follows.
- **Example:** A software company implements a policy requiring all code to pass a static code analysis tool (like SonarQube) and be reviewed by at least one other developer before it can be merged into the main branch. This prevents common coding errors and inconsistencies.

2. Quality Planning (QP)

This involves defining the specific quality objectives and standards for a particular project. It outlines the strategy for how the team will achieve, measure, and control quality throughout the development process.

- **Concept:** Proactively defining quality goals, metrics, and the resources and processes needed to achieve them. This is often documented in a Quality Management Plan.
- **Example:** For a new banking app, the quality plan defines specific metrics such as a target response time for transactions (e.g., under 2

seconds) and a required code test coverage of 80%. It also specifies that user acceptance testing (UAT) will be performed by a representative group of customers before launch.

3. Quality Control (QC)

This is a reactive, product-oriented process that focuses on identifying and correcting defects in the final product. QC activities like testing are performed to verify that the software meets the quality standards established during the planning and assurance phases.

- **Concept:** The set of operational techniques and activities used to check for conformance to quality standards and remove defects.
- **Example:** The QA team executes test cases defined in the quality plan to find bugs and defects. They perform functional tests to ensure all features work as specified and load tests to verify the system can handle expected user traffic. Any defects found are logged in a bug-tracking system like Jira, and the team works to resolve them.

Key concepts across the SDLC

These core concepts are applied throughout the software development life cycle (SDLC) to ensure continuous quality improvement.

Metrics for measuring quality

Metrics provide objective data to track the effectiveness of SQM efforts.

- **Defect Density:** The number of confirmed defects per unit of code (e.g., per thousand lines of code).
    - **Example:** After a feature is tested, the team calculates the defect density. If a new module has a much higher defect density than older, stable modules, it signals a potential quality problem that requires further attention.
- **Mean Time to Recovery (MTTR):** The average time it takes to restore service after a production failure or incident.

    - **Example:** The operations team tracks MTTR to measure the team's incident response effectiveness. A low MTTR indicates that the team can quickly diagnose and fix critical issues in a production environment.

- **Test Coverage:** The percentage of code that is executed by tests.
    - **Example:** A company uses a tool to measure its unit test coverage, finding it is only 50%. The team decides to increase this to 80% to ensure better code quality and fewer defects.

Software Quality Assurance

Software Quality Assurance (SQA) is a systematic process of ensuring that a software product meets its specified quality standards and requirements. It is a proactive, process-oriented discipline that works in parallel with the software development lifecycle to prevent defects from occurring in the first place, in contrast to Quality Control (QC), which is a reactive process focused on finding defects in the finished product.

Core concepts of software quality assurance

1. Process-oriented approach

SQA focuses on improving the development process itself rather than just inspecting the final product. By establishing and refining repeatable, high-quality processes, the likelihood of defects and errors is significantly reduced.

- **Example:** A company implements the Agile methodology, using short development cycles ("sprints"). A key SQA activity is to conduct a process audit at the end of each sprint to identify inefficiencies and improve the workflow. This might reveal that a specific type of bug frequently arises during unit testing, leading the team to add an extra peer review step for that code module.

2. Defect prevention

This is a core principle of SQA, aiming to prevent problems from entering the software early in the development cycle. By identifying and addressing

potential issues at their root cause, costs and development time are significantly reduced.

- **Example:** During the requirements analysis phase, an SQA team reviews the software requirements specification (SRS) document with developers and stakeholders. They identify an ambiguous requirement for handling "invalid user input." To prevent future bugs, the team clarifies the requirement and documents specific validation rules, like setting a maximum character limit and specifying the type of data to be accepted.

3. Continuous improvement

SQA is not a one-time activity but an ongoing process of monitoring, assessing, and refining the software development process. Organizations use models like the Capability Maturity Model Integration (CMMI) to gauge and improve the maturity of their processes.

- **Example:** An SQA team tracks and analyzes bug reports and customer feedback after a new software release. They discover a recurring pattern of usability issues, suggesting the initial design review process was flawed. The team then updates the SQA plan to incorporate usability testing earlier in the design phase, establishing a cycle of continuous improvement.

4. Quality attributes (or factors)

Software quality is defined by a set of measurable characteristics. SQA focuses on ensuring these attributes are achieved throughout the development process. Key attributes include:

- **Reliability:** The software consistently performs as expected without failures.
    - **Example:** An e-commerce application is tested for reliability by simulating high traffic during a major sale event to ensure it doesn't crash or slow down under heavy load.
- **Usability:** The software is easy for users to learn and operate.

    - **Example:** A new mobile banking app is given to a group of first-time users to test the intuitiveness of its navigation and features. Their feedback is used to refine the user interface.

- **Maintainability:** The software can be easily modified, updated, and extended.
    - **Example:** A code inspection is performed to ensure all developers follow the same coding standards and that the code is well-documented. This makes it easier for future developers to understand and modify the code without introducing new errors.
- **Performance efficiency:** The software uses resources like CPU time and memory effectively.
    - **Example:** The development team runs stress tests on a database to see how fast it can retrieve information when handling a large volume of requests. This ensures optimal performance as the user base grows.
- **Security:** The software protects against unauthorized access and data breaches.
    - **Example:** A financial application undergoes regular penetration testing, where ethical hackers attempt to exploit vulnerabilities. This identifies security flaws before malicious actors can.

5. Verification and validation (V&V)

V&V are distinct but complementary activities within SQA.

- **Verification:** Checks if the product is being built correctly. This is often done through reviews, inspections, and walk-throughs to ensure each phase of the development meets the specified requirements.
    - **Example:** During a design review, the SQA team compares the system architecture design against the initial requirements document to verify that all functional needs are being addressed by the design.

- **Validation:** Confirms that the right product was built. It involves testing the software against user expectations and requirements to ensure it meets its intended purpose.
    - **Example:** User Acceptance Testing (UAT) is performed, where end-users test the final product to ensure it meets their expectations and is ready for release.

Formal Technical Reviews (FTRs)

Formal Technical Reviews (FTRs) are a structured and methodical software quality control activity performed by a team of peers to identify defects in software engineering work products. By catching errors early, FTRs prevent defects from advancing to later stages where they become significantly more expensive and difficult to fix.

Core concepts of FTRs

Objectives

The main goal of FTR is to ensure the quality of a software product by finding and removing errors and inconsistencies. Key objectives include:

- **Defect identification:** To uncover errors in functions, logic, or implementation within any software representation, such as design documents or code.
- **Quality assurance:** To confirm compliance with project specifications, requirements, and established standards.
- **Knowledge sharing:** To promote collaboration and a common understanding among team members, which can also train junior engineers.
- **Risk mitigation:** To proactively identify and manage potential issues that could worsen into major risks later in the project.
- **Cost reduction:** To find and fix defects early, which is much cheaper than correcting them during testing or after deployment.

Key principles

For an FTR to be successful, participants must adhere to a core set of guidelines:

- **Review the product, not the producer:** The review should focus on the quality of the work product, not on the individual who created it. The goal is to be constructive, not critical.
- **Set and maintain an agenda:** A structured agenda ensures the meeting stays focused and on track.
- **Limit debate:** The meeting's purpose is to find problems, not solve them. Solutions are discussed and implemented later.
- **Take written notes:** A designated scribe records all issues and points of discussion for follow-up.
- **Insist on advance preparation:** All participants must review the material beforehand to make the meeting productive.
- **Use a checklist:** A prepared checklist helps to focus the reviewers' attention on important issues and ensures consistency.

Participant roles

An FTR involves a small, focused team (typically 3–5 people) with defined roles.

- **Producer (Author):** The person who created the work product. Their role is to clarify points and address the identified defects after the meeting.
- **Moderator:** The facilitator who leads the FTR meeting. They are responsible for keeping the meeting on track, enforcing the rules, and ensuring objectives are met.
- **Reviewers:** The peers and subject matter experts who examine the work product and identify defects.