# Explain Software Testing Principles

**Key Principles**

- **Testing shows the presence of defects, not their absence.** Testing is designed to uncover errors, but even rigorous testing can't guarantee a completely bug-free system. The goal is to find and fix as many defects as possible and reduce risks, not aim for absolute perfection.
    - **Example:** Even after a thorough testing phase, a user might still discover an obscure bug in a less-used feature.
- **Exhaustive testing is not possible.** Testing every possible input combination and execution path is impractical. Testers need to prioritize critical areas and use smart strategies.
    - **Example:** It's impossible to test a simple login form with every conceivable username and password combination.
- **Early testing saves time and cost.** Finding defects early in the software development lifecycle (SDLC) is far more efficient than fixing them after release.
    - **Example:** Identifying a flawed design requirement during the design phase is much cheaper than fixing the same problem in a deployed product.
- **Defect clustering.** Certain areas of the code are more likely to contain defects. Focus testing efforts on these "hotspots" based on past experience and code complexity.
    - **Example:** A new, complex algorithm is more likely to harbor bugs than a well-tested, standard user interface component.
- **Pesticide paradox.** Reusing the same tests repeatedly can make them less effective over time as the software adapts to them. Regularly revise and add new test cases.
    - **Example:** A login button test that only checks with standard credentials might miss issues arising from unusual inputs.
- **Testing is context-dependent.** Different types of software demand different testing approaches. There's no single one-size-fits-all solution.
    - **Example:** Testing a real-time safety-critical system in an aircraft requires different methods and stringency than testing a simple website.
- **Absence of errors fallacy.** Even if no defects are found, it doesn't mean the software perfectly meets user needs or matches specifications.
    - **Example:** Software might pass all its tests but still have a clunky user interface that users find frustrating.

# Explain Software Verification and Validation

**Software Verification**

- **Focus:** Checking if the software conforms to its specifications and requirements. It answers the question: "Are we building the product right?"
- **Activities:**
  - **Reviews:** Examining documents like requirements, design specifications, and code.
  - **Walkthroughs:** Developers guiding stakeholders through the logic of design and code.
  - **Inspections:** Formal, structured process to find defects in work products.
  - **Static Analysis:** Automated tools scanning code for potential issues without executing it.
- **Example:** Verifying that a code module correctly calculates sales tax based on documented tax rules.

**Software Validation**

- **Focus:** Ensuring the software meets the user's needs and intended purpose. It answers the question: "Are we building the right product?"
- **Activities:**
  - **Dynamic Testing:** Executing the software with various inputs. This includes:
    - **Unit Testing:** Testing individual components.
    - **Integration Testing:** Testing how modules work together.
    - **System Testing:** Testing the software as a whole.
    - **Acceptance Testing:** User-focused testing to validate against their requirements.
- **Example:** Conducting user acceptance testing to ensure that a new inventory management system actually meets the needs of warehouse staff.

**Key Differences**

| Feature | Verification | Validation |
|---|---|---|
| Focus | Specifications and design | User needs and intended use |
| Question | "Are we building the product right?" | "Are we building the right product?" |
| Methods | Static (reviews, inspections, analysis) | Dynamic (testing) |
| Timing | Can occur earlier in the SDLC | Occurs after verification is reasonably complete |

**Relationship**

Verification and Validation are complementary, not opposing.

- A system might be verified correctly (meets its specs) but fail validation because the specifications themselves were not aligned with real user needs.
- It's essential to have both strong verification and validation to ensure you are releasing a high-quality software product.

# Explain low level design coupling and cohesion

**Coupling**

- **Definition:** The degree of interdependence between software modules (e.g., classes, functions, components). High coupling means modules are intricately linked, while low coupling implies greater independence.
- **Goal in Low-Level Design:** Strive for low coupling. Loosely coupled modules offer benefits:
    - **Maintainability:** Changes to one module are less likely to have ripple effects throughout the system, making modifications easier.
    - **Reusability:** A module with minimal dependencies can be more easily reused in other projects or within the same system.
    - **Testability:** You can isolate loosely coupled modules for easier unit testing.

**Types of Coupling (from worst to best):**

- **Content Coupling:** One module directly accesses or modifies data in another module.
- **Common Coupling:** Modules share global data.
- **Control Coupling:** One module passes control flags to influence the logic of another.
- **Stamp Coupling:** Modules share parts of a complex data structure (e.g., passing an entire struct when only a few fields are needed).
- **Data Coupling:** Modules primarily share data through parameters.

**Cohesion**

- **Definition:** The degree to which elements within a single module belong together and work towards a single, well-defined purpose. High cohesion means a module is focused and self-contained.
- **Goal in Low-Level Design:** Aim for high cohesion. Highly cohesive modules have advantages:
    - **Understandability:** A well-focused module is easier to comprehend.
    - **Maintainability:** Changes are likely to be localized within the module.
    - **Reusability (potential):** A highly cohesive module encapsulating a specific task might be reusable in other contexts.

**Types of Cohesion (from worst to best):**

- **Coincidental Cohesion:** Module elements are bundled together arbitrarily.
- **Logical Cohesion:** Elements are grouped because they belong to the same logical category, but don't always work together.
- **Temporal Cohesion:** Elements are related because they are used at the same time.
- **Procedural Cohesion:** Elements contribute to a sequence of tasks.
- **Communicational Cohesion:** Elements operate on the same data.
- **Sequential Cohesion:** The output of one element is the input to another.
- **Functional Cohesion:** All elements contribute to a single, clearly defined task. This is the most desirable.

**Relationship Between Coupling and Cohesion**

- **Inverse Relationship:** Generally, as cohesion improves within modules, the coupling between modules decreases. A highly cohesive module tends to be self-contained, minimizing its need to interact with other modules.

# What is UML and different types of UML Diagram

**What is UML?**

- **UML** stands for Unified Modeling Language. It's a standard visual modeling language used in software engineering to create diagrams representing various aspects of a software system.
- **Purpose:** UML provides a way to visualize the structure, behavior, and interactions within a system, helping developers, architects, and stakeholders communicate and understand complex designs.

**Types of UML Diagrams**

UML diagrams fall into two main categories:

**1. Structural Diagrams**

These depict the static elements of a system, such as its components, their relationships, and attributes.

- **Class Diagram:** The foundation of object-oriented modeling. Shows classes in the system, their attributes (data), operations (methods), and how they relate to each other (e.g., inheritance, association).
- **Component Diagram:** Describes how a system is broken down into components and the dependencies between them.
- **Object Diagram:** A snapshot of the system's objects and their relationships at a specific point in time.
- **Deployment Diagram:** Models the physical hardware of a system and how software components are deployed onto it.
- **Package Diagram:** Organizes larger systems by grouping related elements (e.g., classes) into packages.
- **Composite Structure Diagram:** Shows the internal structure of classes and how parts collaborate.

**2. Behavioral Diagrams**

These represent the dynamic aspects of a system, focusing on how objects interact and how the system changes over time.

- **Activity Diagram:** Visually represents a workflow or sequence of activities, including decision points and parallel processes.
- **Use Case Diagram:** Models the functionality of a system from the user's perspective, describing user goals and their interactions with the system.
- **Sequence Diagram:** Depicts the detailed message exchange between objects in a specific scenario, emphasizing the timing and order of interactions.
- **State Machine Diagram (or Statechart Diagram):** Illustrates the different states an object can be in and the events that cause transitions between these states.
- **Communication Diagram:** Similar to sequence diagrams, but emphasizes the structural links between objects involved in the interaction.
- **Interaction Overview Diagram:** A hybrid form, combining aspects of activity diagrams and sequence diagrams for a higher-level view.
- **Timing Diagram:** A specialized diagram used to focus on timing constraints and changes in state over time.

**Benefits of UML**

- **Clear Communication:** Standardized language for sharing complex software designs.
- **System Understanding:** Enhances understanding of system architecture and behavior.
- **Documentation:** Serves as valuable documentation for future reference and maintenance.
- **Design Exploration:** Facilitates the exploration of alternative design choices.

# Explain Test-Case Design

**What is Software Test-Case Design?**

- Test-case design is the process of creating detailed test cases that validate whether software meets its intended requirements and functions correctly.
- A well-designed test case includes the following elements:
    - **Test Case ID:** Unique identifier.
    - **Description:** Brief summary of what the test covers.
    - **Preconditions:** Any conditions that need to be met before executing the test.
    - **Test Steps:** Clear, actionable steps to conduct the test.
    - **Expected Results:** Precisely what the correct outcome should be.
    - **Actual Results:** Space to document observations during testing.
    - **Status:** E.g., Pass, Fail, Blocked.

**Test-Case Design Techniques**

There are several common strategies to design effective test cases:

- **Equivalence Partitioning:** Dividing input data into groups that should produce the same behavior (valid and invalid). You test a representative from each group.
- **Boundary Value Analysis:** Focuses on input values at the edges of those equivalence partitions (e.g., minimum/maximum values, just below/above).
- **Decision Table Testing:** Describes combinations of inputs and expected results for scenarios with multiple conditions.
- **State Transition Testing:** Validating how the system reacts to events based on its current state (think of state transition diagrams as a basis).
- **Error Guessing:** Using experience and intuition to design tests for areas where errors might be more likely to occur.

**Additional Considerations**

- **Requirements Coverage:** Ensure your test cases address all functional and non-functional requirements.
- **Positive and Negative Testing:** Include tests that validate both expected behavior and how the system handles incorrect inputs or conditions.
- **Regression Testing:** Consider how new features or changes might impact existing functionality.
- **Test Levels:** Design test cases for different testing stages like unit testing, integration testing, and system testing.

**Example: Test Case for a Login Feature**

| Test Case ID | Description | Preconditions | Test Steps | Expected Result | Actual Result | Status |
|---|---|---|---|---|---|---|
| TC01 | Valid Login | User account exists | 1. Enter valid username <br> 2. Enter valid password <br> 3. Click "Login" | User is logged in and redirected to the dashboard | | |

**Best Practices**

- **Clear and Concise:** Write test cases that are easy to understand and execute.
- **Version Control:** Manage test cases in a version control system to track changes.
- **Prioritize:** Focus on high-risk and critical areas of the application.
- **Regular Review:** Periodically review and update test cases as the software evolves.

**Tools**

Test management tools like TestRail and Jira can aid in organizing, writing, and executing test cases.

# Explain Lines of Code

**What is SLOC?**

- Lines of Code (LOC) is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.
- SLOC is a simple and intuitive measure, but it has its limitations and should be used with caution.

**Types of Lines Counted**

- **Physical SLOC:** Counts any line of code with content, regardless of its purpose.
- **Logical SLOC:** Counts individual executable statements. Multiple statements on a single line are counted separately.

**What Is Not Included**

- **Blank lines:** Empty lines are ignored.
- **Comments:** Explanatory comments are not counted.
- **Header lines:** Some definitions include header lines, others do not – the distinction is important.

**Potential Use Cases**

- **Rough Project Size Estimate:** SLOC can provide a very basic initial idea of the scale of a project.
- **Productivity Comparisons (WITH CAUTION):** Can be used to make broad comparisons between projects written in the same language and using similar coding styles.
- **Benchmarking:** Tracking changes in SLOC over time can reveal trends in the size of a codebase during development.

**Major Limitations of SLOC**

- **Language-Dependent:** An algorithm written in a language like Python vs. Java will have vastly different SLOC due to language paradigms. Comparisons only make sense within the same language.
- **Coding Style Influence:** Programmers with different spacing and formatting styles can greatly influence SLOC for the same functionality.
- **Does Not Reflect Complexity:** Complex logic might be implemented concisely, while simple logic could take many lines. SLOC alone says nothing about complexity.
- **Doesn't Indicate Quality:** More lines of code do not necessarily equal better software.

**Better Alternatives**

- **Function Point Analysis:** A more sophisticated metric based on the complexity and number of user functions within the software.
- **Cyclomatic Complexity:** Measures the number of linearly independent paths through the code, correlating with difficulty in testing and maintaining code.
- **Focus on Quality and Functionality:** Emphasizing whether the software meets requirements, is well-designed, and is efficient, rather than just the raw code count.

**In Summary**

SLOC is a historical metric. While somewhat intuitive, it's important to understand its limitations. SLOC should never be the sole measure for assessing a project's size, development effort, or quality.