SE Module 3

13-Module 3_ Strategic Approach to Software Testing,-03-09-2024

What is Software Testing?

- Software testing is essential to find errors in a program.
- Testing ensures that a program meets its requirements and works as expected, preventing issues that could affect users.
- Testing is a planned activity involving systematic steps, strategy, and clear results.

Definitions of Software Testing:

- **Myer's Definition**: Executing a program to find errors.
- **Dijkstra's View**: Testing can show the presence of bugs but never prove they are absent.

Why Software Testing is Needed:

- To identify and fix errors.
- To ensure the system meets the requirements and works in its intended environment.
- To improve software quality, making it bug-free, reliable, and user-friendly.

Common Software Problems:

- Incorrect calculations, data handling, and processing.
- Usability issues, inconsistent processing, and poor performance.
- Errors in business rules and system interfaces.

Importance of Testing:

- To catch defects early and avoid customer-facing issues.
- Provides confidence that the software meets its purpose.

- Real-world examples show the serious consequences of failing to test, such as software glitches causing crashes in financial systems or aircraft.

Testing Challenges:

- Time pressures, changing requirements, and complexity of software often lead to mistakes.
- Manual testing and inexperienced testers are common issues.

Benefits of Software Testing:

1. **Cost-Effective**: Finding bugs early saves money in the long run.
2. **Security**: Reduces risks by removing vulnerabilities early.
3. **Product Quality**: Ensures a quality product for customers.
4. **Customer Satisfaction**: Guarantees a better user experience through UI/UX testing.

Skills Required for a Software Tester:

- Good verbal and written communication.
- Analytical skills and problem-solving ability.
- Time management and organization skills.
- Passion for software quality and attention to detail.

Testing is essential to deliver quality, reliable, and secure software, ensuring customer satisfaction and minimizing risks.

14-Strategic Issues-05-09-2024

What is Software Testing?

Software testing is the process of evaluating a software application to find and fix errors. These errors may have been accidentally introduced during the design or development stages. Testing ensures that the software works as expected.

Who Does Software Testing?

- **Project Manager**: Responsible for planning the testing strategy.
- **Software Engineers**: Help develop the tests.
- **Testing Specialists**: Focus on executing tests and finding defects.

Why is Software Testing Important?

Testing is crucial because it:

- Accounts for a large portion of the project’s time and effort.
- Ensures that the software works as expected and meets the requirements.
- Helps avoid costly mistakes by identifying errors early.

What are the Steps in Software Testing?

1. **Start small**: Testing begins with individual components (unit testing).
2. **Expand gradually**: After units are tested, combine them and test their interactions (integration testing).
3. **End big**: Finally, the whole system is tested (system testing).

Work Product in Testing:

The primary work product of testing is the **Test Specification Document**, which outlines:

- The types of tests to be done.
- The conditions to be tested.
- Expected outcomes for each test.

How to Ensure Proper Testing?

Before starting tests, review the **Test Specification** to make sure it’s complete and includes all the necessary test cases.

Strategic Approach to Software Testing:

- **Testing must be planned** to avoid wasting time and resources.
- **Systematic process**: Testing should follow a structured approach, ensuring that each phase is covered thoroughly.
- **Types of testing are applied at different stages** of software development, like testing individual units (unit testing), combining units (integration testing), and testing the full system (system testing).
- **Developers and independent testers both test**. Developers do unit testing, while independent testers usually focus on finding more complex issues.

Types of Software Testing Strategies:

There are several strategies that testers use depending on the situation. Here are the key ones:

1. **Analytic Testing Strategy**: This is about using analysis and reasoning to understand the software's requirements and then creating test cases. It's like studying blueprints before building a house.
2. **Philosophical Testing Strategy**: This focuses on the overall principles of testing. It's about having the right mindset and approach to testing.
3. **Model-based Testing Strategy**: In this strategy, models (representations of how the software should behave) are used to generate test cases. Think of using a scale model to plan out how a house will be built.
4. **Dynamic Testing Strategy**: This tests the software by actually running it. It's like walking through a house to check if everything works (doors open smoothly, floors don't creak, etc.).
5. **Methodical Testing Strategy**: It follows a detailed checklist to ensure everything gets tested systematically. This is like inspecting a house with a list of things to check, like plumbing and electrical systems.
6. **Process-oriented Testing Strategy**: This strategy focuses on making sure the testing process is efficient and follows industry standards. It's about making sure the process itself works smoothly.

Verification and Validation:

- **Verification**: This checks if the software is built correctly, according to its specifications. It ensures that the product is being developed as expected (internal check).
- **Validation**: This checks if the software meets the user's needs and if it solves the problem it was designed for. It’s about making sure the software is the right product (external check).

Who Tests the Software?

- **Developer Testing**: Developers usually test the software while they are developing it. They know the system well but might not test it thoroughly since they are focused on completing the code.
- **Independent Tester**: These testers focus solely on the quality of the software and are separate from the development team. They approach testing with a fresh perspective and try to break the system by finding issues others might miss.

Strategic Issues in Testing:

1. **Clear Objectives**: State the testing goals upfront so everyone knows what they’re trying to achieve.
2. **User Profiles**: Understand who will use the software and test according to their needs.
3. **Rapid Testing Cycles**: Plan to test frequently and quickly, adjusting as the software evolves.
4. **Robust Software Design**: Design software in a way that it can test itself to catch errors automatically.
5. **Formal Reviews**: Conduct technical reviews before starting testing to catch issues early.
6. **Continuous Improvement**: Always look for ways to improve the testing process, making it more efficient over time.

Unit Testing:

Unit testing is the process of testing individual parts or components of the software, typically during the development phase.

- **Purpose**: Ensure that each part of the software works as expected on its own.
- **Who does it?**: Developers usually perform unit tests.
- **Why perform unit testing?**
  - It helps catch bugs early, saving costs.
  - It helps developers understand and reuse code.
  - It makes debugging easier since issues are isolated to smaller parts of the system.
- **Disadvantages**:
  - It might not catch issues that arise when parts of the system interact with each other.
  - Requires writing extra code for testing, which can be time-consuming.

Module Testing:

Module testing focuses on testing software modules (or small parts) that have already been coded.

- **Difference from Unit Testing**:
  - Unit testing focuses on isolated units, while module testing tests multiple units combined to see how they interact.
  - Module testing is usually done after unit testing, once units are integrated into modules.

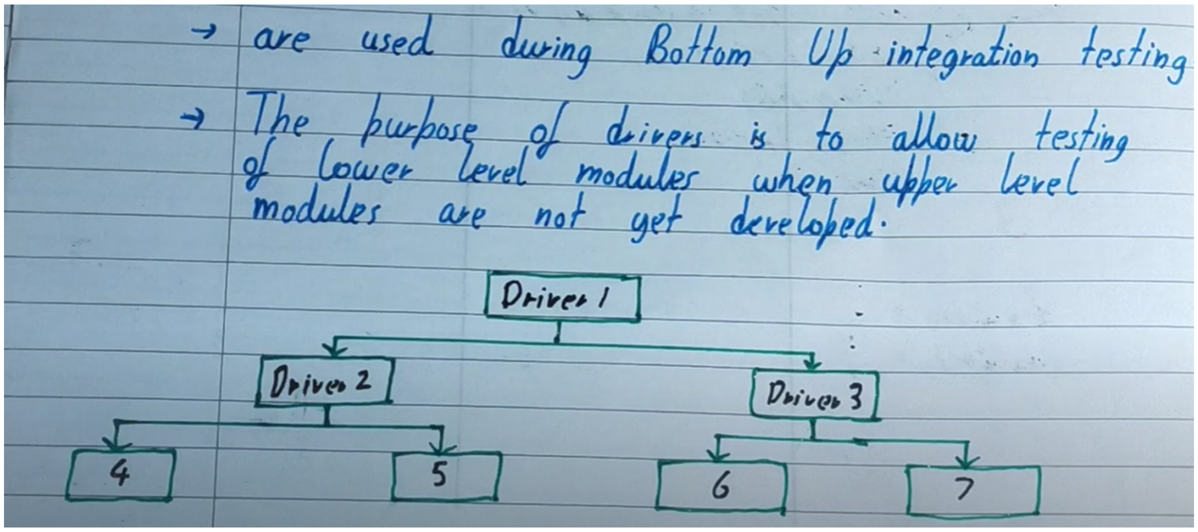
Integration Testing:

Integration testing is done to check if different software components (units or modules) work together when combined.

- **Why do we do it?**:
  - To check if changes in one module affect others.

- To ensure different modules interact correctly and to catch issues missed in unit testing.

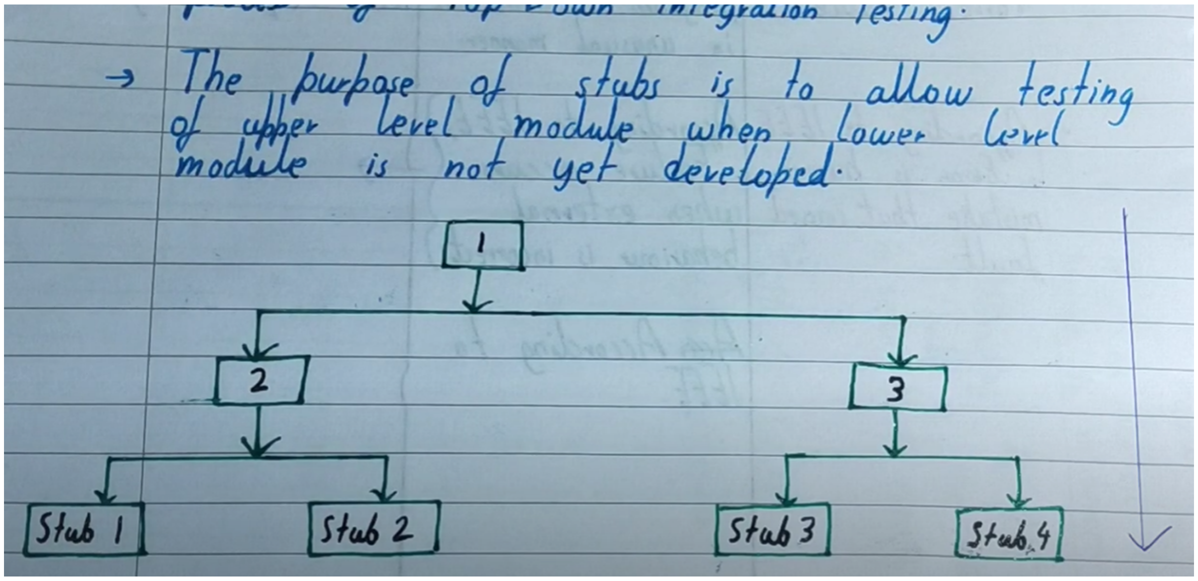
- **Test Drivers and Stubs**:
  - **Drivers**: Programs used to test a module by calling its functions. In Bottom-Up Testing Approach Drivers are used.



→ are used during Bottom Up integration testing

→ The purpose of drivers is to allow testing of lower level modules when upper level modules are not yet developed.

  - **Stubs**: Dummy functions used when a module is still being developed or isn't available. In Top-Down Testing Approach Stubs are used.



Top Down integration testing.

→ The purpose of stubs is to allow testing of upper level module when lower level module is not yet developed.

Approaches to Integration Testing:

1. **Big Bang Integration Testing**:
    - All modules are integrated at once and tested together.
    - **Advantages**: Simple for small systems.
    - **Disadvantages**: Hard to isolate issues because everything is tested at once.
2. **Incremental Integration Testing**:
    - Modules are added and tested one by one.
    - **Advantages**: Easier to identify issues because only a few modules are integrated at a time.
    - **Types**:
        - **Top-Down Testing**: Start testing from the top modules and move downward. Test stubs are used for lower-level modules.
        - **Bottom-Up Testing**: Start from the lowest-level modules and work upward.

Incremental Testing Approach

Incremental testing is when we test parts of the system as they are built, step by step. The idea is to start with smaller, manageable parts and gradually build towards testing the entire system.

---

1. Top-down Testing Approach

- **What is it?**

  This testing approach starts with the **top-level** module (or high-level components) and works downward, adding and testing lower-level modules one by one.

- **Advantages of Top-down Testing**:
    - **Easy fault localization**: Errors are easier to find because you're testing from the top.

    - **Quick prototype design**: You can quickly design and test high-level features.
    - **Prioritize critical modules**: Start testing the most important parts first.
    - **Fix design flaws quickly**: It's easier to identify and fix design problems in early stages.
- **Drawbacks of Top-down Testing**:
    - **Not testing lower-level modules**: Since you start from the top, lower-level modules are not tested until later, which might delay finding issues there.

---

2. Bottom-up Testing Approach

- **What is it?**

  In this approach, we start testing from the **lowest-level** modules and work our way upwards in the system's architecture.
- **Advantages of Bottom-up Testing**:
    - **Easy to test and develop**: Since the foundation (low-level modules) is tested first, the process is more stable.
    - **Efficient integration testing**: It's easy to see how lower-level modules interact.
    - **Simple to create test conditions**: You can build tests for small parts before connecting them to larger ones.
- **Drawbacks of Bottom-up Testing**:
    - **System-level functions not visible**: Testers can't check the overall system functionality until higher modules are tested.

---

3. Sandwich Testing Approach (Hybrid Integration Testing)

- **What is it?**

  The sandwich approach combines both **Top-down** and **Bottom-up** strategies. It tests the top and bottom modules simultaneously, working towards the middle of the system.

- **Advantages of Sandwich Testing**:
  - **Useful for large projects**: Especially beneficial for large systems with many sub-projects.
  - **Parallel testing**: The top and bottom parts can be tested at the same time.
  - **Time-saving**: It saves time as you test multiple layers at once.
  - **Better coverage**: It gives more coverage with the same set of stubs and drivers.
- **Drawbacks of Sandwich Testing**:
  - **Costly**: It can be expensive to manage and implement.
  - **Not for highly interdependent systems**: Works best when modules are loosely connected.
  - **High need for stubs and drivers**: You need more dummy modules to simulate the system.

Stubs and Drivers in Software Testing

- **Stub**: A stub is a placeholder (or dummy) function that simulates the behavior of a module that the module under test depends on.
  - **Used in Top-down testing**.
  - **Example**: If a module A needs to interact with module B, but module B is not ready yet, module B's functionality is replaced with a stub.
- **Driver**: A driver is a dummy function that simulates a higher-level module (one that calls the module under test).
  - **Used in Bottom-up testing**.
  - **Example**: If module A needs to call module B but module A is not ready, a driver is used to simulate module A's functionality.

Regression Testing

- **What is it?**

Regression testing ensures that changes in the software (like new features or bug fixes) do not negatively affect existing functionality.

- **When is it needed?**
  - After a code change or bug fix.
  - When new features are added.
  - After performance optimizations.
- **Types of Regression Testing**:
  - **Unit Regression**: Testing after a small change, focusing on the affected unit.
  - **Partial Regression**: Testing changes in specific parts of the system.
  - **Complete Regression**: Testing the entire system after changes.
- **Advantages of Regression Testing**:
  - **Ensures quality**: It helps maintain the stability and reliability of the system.
  - **Can be automated**: You can use tools to automate and speed up the testing.
  - **Helps in preventing repeated issues**: Any previously fixed bugs are less likely to appear again.
- **Disadvantages of Regression Testing**:
  - **Time-consuming**: Especially if done manually.
  - **Small changes can require extensive retesting**: Even small code updates may require running the full set of tests.

Smoke Testing

- **What is it?**

  Smoke testing is a quick test done on initial builds to check if the critical functionalities of the software are working. It is also called **Build Verification Testing**.
- **Purpose**:

  - Ensure the basic functions of the software are stable before moving on to more detailed testing.
  - Quickly identify major issues in the build.

Sanity Testing

- **What is it?**

  Sanity testing is a **subset of regression testing**. It checks that the changes made to the software (like bug fixes or new features) are working as expected.

- **Purpose**:
  - Verify if the software is stable enough to proceed with further, more detailed testing.

Object-Oriented Testing

- **What is it?**

  Object-oriented testing focuses on testing software designed using object-oriented programming principles (e.g., classes and objects).

- **Unit Testing in OO Context**:
  - Instead of testing algorithms or modules, you test classes in isolation, focusing on the class's operations and state.
- **Integration Testing in OO Context**:
  - **Thread-based testing**: Tests interactions between classes in client-server applications, treating each class interaction as a thread.
  - **Use-based testing**: Focuses on testing complete system transactions, covering all aspects of system use from start to finish.

Testing Web Applications

- **Types of Testing for Web Applications**:
  - **Functionality Testing**: Check if the web app's features work as expected.
  - **Usability Testing**: Ensure the app is easy to use for end-users.

    - **Interface Testing**: Test the interaction between the web app and its components (e.g., databases).
    - **Compatibility Testing**: Ensure the app works across different browsers, operating systems, and devices.
    - **Performance Testing**: Assess the app's speed, responsiveness, and stability under load.
    - **Security Testing**: Test the web app’s defenses against unauthorized access or attacks.
- **Strategies**:
    - Review content, design, and functionality for errors.
    - Test navigation, user interface, and overall experience.
    - Test the app under different environments to ensure compatibility and security.

Validation Testing

- **What is it?**

  Validation testing ensures the software meets the user’s needs and performs as expected, checking if the features are working correctly.
- **Validation-Test Criteria**:
    - **Pass**: The software works as required.
    - **Fail**: There’s a deviation, and a deficiency list is created to track issues that need fixing.
- **Configuration Review (Audit)**:
    - Ensures all components are correctly created, documented, and ready for support and future updates.
- **Alpha and Beta Testing**:
    - **Alpha Testing**: Performed by internal testers to catch bugs before release.

    - **Beta Testing**: Done by actual users in a real environment to see how the software performs.

System Testing

System testing is an overall evaluation of the entire system to verify that everything works together.

- **Types of System Testing**:
    - **Recovery Testing**: Ensures the system can recover after failures.
    - **Security Testing**: Verifies the system is protected from unauthorized access.
    - **Stress Testing**: Tests how the system performs under extreme conditions (e.g., high load).
    - **Performance Testing**: Measures the system's performance under normal and peak conditions.

Summary

By understanding each of these testing approaches, strategies, and methodologies, you'll be prepared to tackle questions about them in your exam. Focus on the key differences between approaches (like Top-down vs. Bottom-up), the purpose and types of testing (e.g., regression vs. smoke testing), and the goals of validation and system testing. Each type of test has its purpose in ensuring the software meets its requirements and works as expected in real-world use.

Just remember:

- Testing strategies vary depending on the phase of development and the needs of the software.
- Unit and module testing focus on individual parts, while integration testing checks how parts work together.
- Verification ensures you're building the software right, and validation ensures you're building the right software.

15-Test Strategies for Conventional Software-10-09-2024

Debugging:

What is Debugging?

- Debugging is the process of finding and fixing errors (bugs) in a program after testing.
- It happens after successful testing when bugs are identified.

Origin of Debugging:

- The term "debugging" comes from Grace Hopper’s discovery in 1945 when a moth caused a malfunction in a computer’s relay.

The Debugging Process:

- **Two Outcomes**: Either the cause is found and fixed, or it remains elusive. If the cause isn’t clear, testers create new test cases to confirm suspicions.
- Debugging is different from testing: Testing identifies mistakes, while debugging fixes them.

Challenges of Debugging:

- It can be frustrating because bugs can be complex, hard to trace, or caused by multiple factors.
- Symptoms might be random, may disappear temporarily, or seem to result from non-errors.
- Timing issues and human error can also complicate debugging.

Debugging Strategies:

- **Brute Force**: Trying all possibilities, which is often inefficient.
- **Backtracking**: Working backward from where the error appeared.
- **Induction & Deduction**: Using logic to narrow down the potential causes through binary partitioning (splitting problems in half).

Correcting the Error:

- Once the cause is identified, ask three key questions before fixing:
    1. **Is the cause present elsewhere in the program?**
    2. **What other issues might this fix create?**
    3. **What could have been done to prevent the bug initially?**

Why Debugging is Difficult:

- It involves problem-solving and can cause frustration due to the difficulty of finding connections between symptoms and causes, especially with complex bugs or random errors.

16-Fundamentals, Black box Testing-12-09-2024

Software Testing Overview:

- **Testing Perspectives**:
    1. **White-box Testing**: Focuses on the internal logic and structure of the program. Tests are derived from the code.
    2. **Black-box Testing**: Focuses on the functionality and behavior of the software, testing based on user requirements.
- **Test Case Design**:
    - **White-box Techniques**: Includes **path coverage**, **statement coverage**, **branch coverage**, and **basis path testing**.
    - **Black-box Techniques**: Includes **equivalence partitioning**, **boundary value analysis**, **decision table testing**, **state transition testing**, **error guessing**, **graph-based testing**, and **comparison testing**.

Software Testing Fundamentals:

- **Testability**: Refers to how easily software can be tested. Key factors include:
    - **Operability**: Better functioning makes testing easier.
    - **Observability**: The more visible the software's operations, the easier it is to test.

- **Controllability**: Better control of software improves test automation and efficiency.
- **Simplicity**: Simpler software is easier to test.
- **Stability**: Fewer changes mean fewer disruptions during testing.
- **Understandability**: The more we know, the smarter the testing.

White-box Testing:

- **White-box Testing (Structural)**: Focuses on program code. It includes:
  - **Path Coverage**: Tests all possible paths in the program.
  - **Statement Coverage**: Tests every individual statement in the program.
  - **Branch Coverage**: Ensures all branches in the program are tested.
- **Basis Path Testing**: Identifies the set of independent paths through the program, ensuring all statements are executed at least once.
  - **Cyclomatic Complexity**: Measures program complexity by calculating independent paths (using the formula **V(G) = E - N + 2** where E = edges, N = nodes).

Cyclomatic Complexity:

- **Cyclomatic Complexity (CC)**: Determines the complexity of a program based on decision points (like `if` statements, loops).
  - Example: A program with 1 decision point has a complexity of **CC = 2**.
  - **Test Cases**: The number of independent paths equals the complexity, guiding the number of test cases.
  - **Interpretation**:
    - CC = 1-10: Simple, structured code (low effort).
    - CC = 10-20: Medium complexity (medium effort).
    - CC > 40: Very complex, hard to test.

- **Independent Paths**: Unique execution paths that cover new processing steps or conditions.

Black-box Testing:

- **Black-box Testing**: Focuses on testing the software's functionality without knowledge of its internal code.
  - **Functional Testing**: Includes smoke testing, sanity testing, integration testing, system testing, regression testing, and user acceptance testing.
  - **Non-functional Testing**: Includes usability testing, load testing, performance testing, compatibility testing, stress testing, and scalability testing.
- **Techniques**:
  - **Equivalence Partitioning**: Divides input into valid and invalid partitions to reduce the number of test cases.
  - **Boundary Value Analysis**: Tests values at the extreme ends of input ranges.
  - **Decision Table Testing**: A table representing inputs vs. outcomes to check all possible combinations.
  - **State Transition Testing**: Tests software states based on events.
  - **Error Guessing**: Uses tester experience to identify error-prone areas.
  - **Graph-based Testing**: Identifies object relationships and tests based on flow.

Loop Testing:

- **Loop Testing**: Focuses on testing loops in the program.
  - **Types**:
    - **Simple Loop**: Tests different numbers of passes through a loop.
    - **Nested Loop**: Tests combinations of inner and outer loops.
    - **Concatenated Loop**: Tests multiple independent loops.

        - **Unstructured Loop**: Combines multiple loops in no specific order.
    - **Why Loop Testing**: Helps identify loop issues like uninitialized variables and performance bottlenecks.

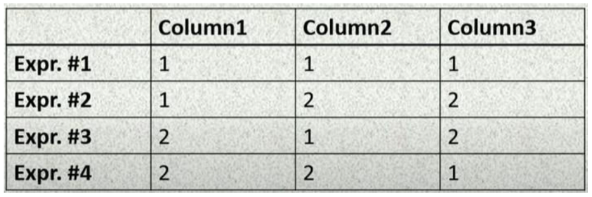
Comparison Testing:

- **Comparison Testing**: Compares your software with competitors to find strengths and weaknesses.
    - **When to Perform**:
        - Early stages to catch requirements/design flaws.
        - Mid-stage to compare UI and functionality.
        - Late-stage to compare quality, speed, and performance.

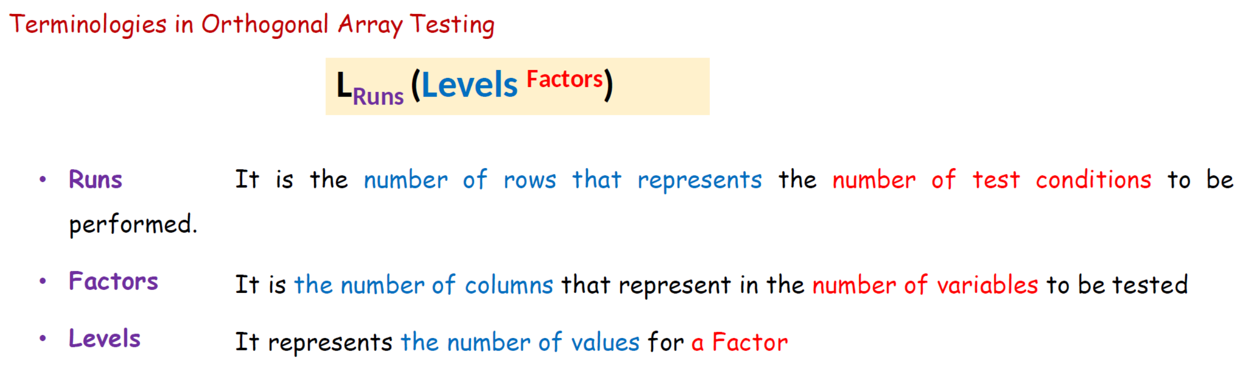
Orthogonal Array Testing (OAT):

- **OAT**: A statistical method that tests pair-wise combinations of inputs.
    - **Benefits**: Reduces the number of test cases while still covering most combinations.
    - **Example**: If a webpage has three sections (Top, Left, Right) with two states (Visible or Hidden), OAT helps test fewer combinations efficiently.

Orthogonal Array Testing



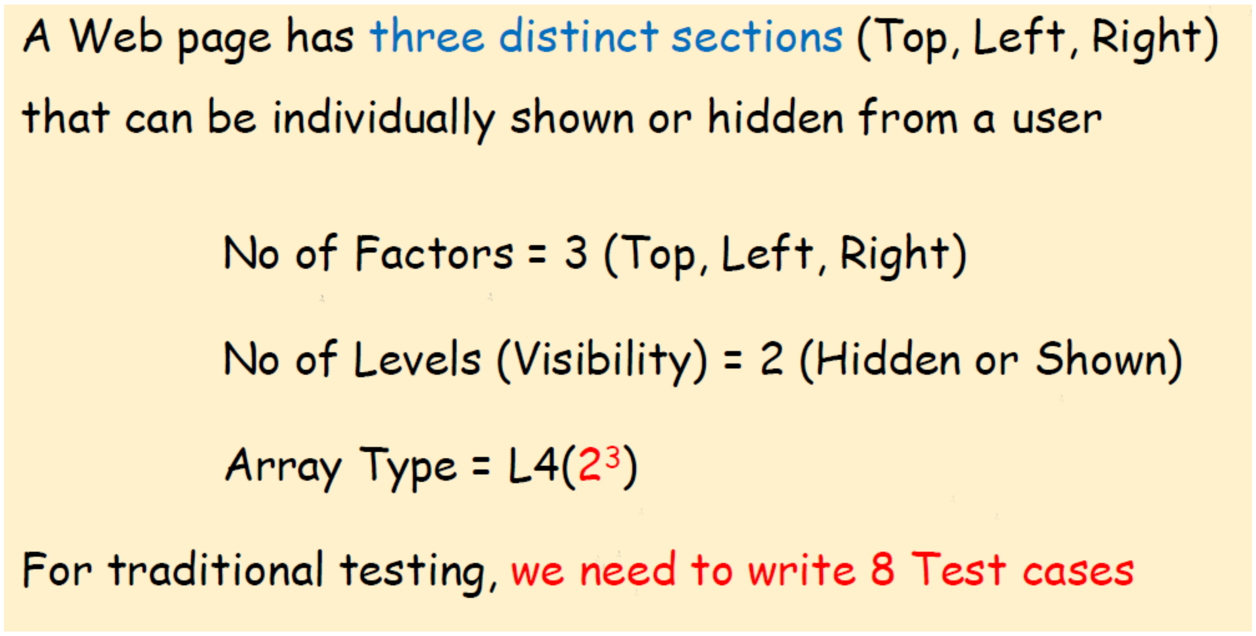
	Column1	Column2	Column3
Expr. #1	1	1	1
Expr. #2	1	2	2
Expr. #3	2	1	2
Expr. #4	2	2	1



Terminologies in Orthogonal Array Testing

L_{Runs} (Levels ^{Factors})

- **Runs** It is the number of rows that represents the number of test conditions to be performed.
- **Factors** It is the number of columns that represent in the number of variables to be tested
- **Levels** It represents the number of values for a Factor



A Web page has three distinct sections (Top, Left, Right) that can be individually shown or hidden from a user

No of Factors = 3 (Top, Left, Right)

No of Levels (Visibility) = 2 (Hidden or Shown)

Array Type = $L4(2^3)$

For traditional testing, we need to write 8 Test cases

Model-based Testing (MBT):

- **MBT**: Uses models (e.g., UML diagrams) to define expected system behaviors and test transitions between system states.
  - Steps:

1. Create a behavioral model.
2. Identify inputs that trigger state changes.
3. Define expected outputs.
4. Execute and compare results.

Key Differences Between White-box and Black-box Testing:

- **White-box Testing**:
  - Requires knowledge of the code.
  - Focuses on testing the internal logic (e.g., unit testing).
  - Done by developers or testers with programming knowledge.
- **Black-box Testing**:
  - Focuses on testing functionality without knowledge of the code.
  - Done by testers who focus on requirements and specifications.
