TESTING TECHNIQUES

Mansi Thakur

Table of Contents

1. Introduction to Testing Techniques

Testing techniques are systematic methods used to design, execute, and manage test cases to ensure software quality. These techniques help identify defects, validate functionality, and optimize the testing process. They are an integral part of both manual and automated testing efforts.

2. Purpose and Importance of Testing Techniques

Structured Approach: Testing techniques provide a structured way to identify test cases and scenarios.
Coverage Optimization: They ensure optimal test coverage with minimal effort.
Defect Detection: Enable identification of both functional and non-functional defects.
Efficiency: Reduce redundancy in test cases, saving time and resources.
Early Defect Prevention: Techniques like static testing detect defects early, reducing cost and effort in later stages.

3. Classification of Testing Techniques

Testing techniques are classified into **Static** and **Dynamic** approaches, each serving specific purposes in the software testing lifecycle.

1. **Static Techniques**

Evaluate software artifacts (e.g., requirements, design, code) without execution.
Focused on defect prevention.

2. **Dynamic Techniques**

Involve executing the software to validate its behavior.
Focused on defect detection. Includes:

- **Black Box Testing Techniques**: Based on functionality/specification.
- **White Box Testing Techniques**: Based on internal structure.
- **Experience-Based Testing Techniques**: Based on tester insights.

4. Static Testing Techniques

Static testing is performed early in the development lifecycle to catch defects before the software is executed.

4.1 Reviews

Reviews involve examining software artifacts collaboratively.

1. **Informal Reviews**
   - Used for initial feedback or brainstorming or discussions among team members.
2. **Walkthroughs**
   - The author presents the document or code to peers for feedback.
   - **Objective**: Identify gaps and ensure understanding.
3. **Technical Reviews**
   - A structured review involving technical experts.
   - **Focus**: Detect technical errors, logical inconsistencies.
4. **Inspections**
   - The most formal review process.
   - Includes a moderator, defined roles, and checklists.
   - **Outcome:** Detailed defect reports.

4.2 Static Analysis

Static analysis tools automate the examination of code and models.

1. **Code Analysis**
   - Focus: Syntax errors, security vulnerabilities, adherence to standards.
   - Example Tools: SonarQube, Coverity.
2. **Model Checking**
   - Ensures models (e.g., UML diagrams) adhere to consistency rules.

5. Dynamic Testing Techniques

Dynamic testing validates the functionality, performance, and usability of the software during execution.

5.1 Black Box Testing Techniques/Specification Based Testing

Black Box Testing examines the software from the user's perspective. Testers interact with the software by providing inputs and verifying the outputs without knowing how the software processes these inputs internally. Characteristics of Black Box Testing

- **No Knowledge of Internal Code**: Testers are unaware of the implementation, code structure, or architecture.
- **Focus on Inputs and Outputs**: Validates whether the software behaves as expected for given inputs.
- **Requirement-Based**: Relies on user stories, specifications, or requirements.
- **Applicable at All Levels**: Can be used in unit, integration, system, and acceptance testing.

A) When to Perform Black Box Testing?

Black Box Testing is performed in the following scenarios:

- **Requirement Validation**: Ensuring the system behaves as per the documented requirements.
- **Functional Testing**: Testing the functional aspects of the system.
- **User Acceptance Testing (UAT)**: Verifying that the software meets the end-user expectations.
- **Regression Testing**: Ensuring that new changes do not affect existing functionalities.

B) How to Perform Black Box Testing?

- **Understand Requirements**: Review functional requirements, user stories, or specifications.
- **Identify Test Scenarios**: Derive scenarios covering all functional aspects of the application.
- **Design Test Cases**: Use techniques like EP, BVA, and Decision Table Testing.
- **Set Up the Environment**: Prepare the test environment, including test data and tools.
- **Execute Tests**: Provide inputs, observe outputs, and compare them against expected results.
- **Log Defects**: Document any mismatches or defects for resolution.
- **Retest and Validate**: Verify the fixes after defect resolution.

C) Tools for Black Box Testing

- **Selenium**: Automates web-based application testing.
- **Postman**: API testing and validation.
- **JMeter**: Performance testing tool.
- **Cypress**: Modern front-end testing framework.
- **TestComplete**: Comprehensive functional testing tool.
- **SoapUI**: Web service testing.

D) Advantages of Black Box Testing

- **No Need for Coding Knowledge**: Suitable for testers with minimal technical expertise.
- **User-Centric**: Tests the system from an end-user perspective.
- **Wide Applicability**: Can be applied to any software level (unit, integration, system).
- **Effective for Large Systems**: Simplifies testing by focusing on inputs and outputs.

E) Advantages of Black Box Testing

- **Limited Coverage**: Cannot uncover issues in the internal structure or logic.
- **Dependency on Documentation**: Relies heavily on well-defined requirements and specifications.
- **Redundant Test Cases**: May create overlapping test cases due to lack of insight into the internal code.
- **Difficult Debugging**: Identifies defects but does not pinpoint their root cause.

F) Best Practices for Black Box Testing

- **Understand Requirements Thoroughly**: Ensure clarity on functional and non-functional requirements.
- **Prioritize Scenarios**: Focus on critical and frequently used functionalities.
- **Combine Techniques**: Use EP, BVA, and Decision Tables for comprehensive coverage.
- **Automate Where Possible**: Leverage tools for repetitive tasks like regression testing.
- **Collaborate with Developers**: Gain insights into the system's behavior for better test design.

5.2 Types of Black Box Testing

A) Equivalence Partitioning (EP)

Equivalence Partitioning divides input data into logical partitions or "equivalence classes." Within each partition, test cases are expected to yield similar results.

Purpose:

To reduce the total number of test cases by ensuring that one test case from a partition is sufficient to represent all other values within that group.

Steps to Perform:

- Analyze the input domain and identify partitions (valid and invalid ranges).
- Select representative values from each partition.
- Design test cases using these representative values.

Tools for EP:

- Test Management Tools like TestRail or qTest.

Advantages:

- Reduces the number of test cases significantly.
- Ensures coverage of all equivalence classes.

Disadvantages:

- Does not test boundary conditions (handled by BVA).

How to Perform:

1. Identify all possible inputs for a given functionality.
2. Group inputs into valid and invalid partitions based on expected behavior.
3. Select one representative value from each partition for testing.

Example:

For an age input field that accepts values between 18 and 60:

- Valid partitions: 18–60
- Invalid partitions: <18 and >60

  Test cases:

- Valid: 25, 45
- Invalid: 17, 61

When to Use:
When a functionality accepts a range or set of values.

B) Boundary Value Analysis (BVA)

Boundary Value Analysis focuses on the edges of input ranges where errors are most likely to occur.

Purpose:
To test values at, just above, and just below the boundaries of equivalence partitions.

How to Perform:

- Identify boundaries for input ranges.
- Create test cases for:
  - Lower boundary (min - 1, min, min + 1)
  - Upper boundary (max - 1, max, max + 1)

Example:
For an input field accepting 18–60:
Test cases: 17, 18, 19, 59, 60, 61

When to Use:
When dealing with input ranges or limit conditions.

Advantages:

- Detects boundary-related defects.
- Increases the likelihood of finding edge-case issues.

Limitations:

- Focused only on boundaries; may miss issues within partitions.

C) Decision Table Testing

A tabular representation of inputs, conditions, and expected outcomes to test complex decision-making logic.

Purpose:
To validate the system's response to combinations of inputs or rules.

How to Perform:

- Identify all conditions and actions for a feature.
- Construct a decision table mapping input combinations to expected outputs.
- Create test cases for each rule in the table.

Example for a discount system:
Conditions: Membership (Yes/No), Purchase Amount (Above $100/Below $100)

- Actions: Apply 10% Discount, No Discount

Membership	Amount > $100	Discount
Yes	Yes	10%
Yes	No	None
No	Yes	None
No	No	None

When to Use:
For systems with complex business rules or logic.

Advantages:

- Ensures all input combinations are covered.
- Clearly maps inputs to expected results.

Limitations:

- Can become complex for systems with numerous conditions.

D) State Transition Testing

State Transition Testing focuses on the system's behavior when transitioning between different states due to events or inputs.

Purpose:

To ensure the system behaves correctly for valid transitions and gracefully handles invalid transitions.

How to Perform:

- Identify states and events causing transitions.
- Create a state transition diagram or table.
- Design test cases to validate all transitions.

Example:

Testing a login system:

- States: Logged Out → Logging In → Logged In → Logging Out
- Events: Entering credentials, timeout, clicking "Logout"

When to Use:

For applications like workflows, user sessions, or devices with finite states.

Advantages:

- Ideal for systems with state-dependent behavior.
- Helps identify invalid state transitions.

Limitations:

- Complex to implement for systems with many states.

E) Use Case Testing

Testing based on user interactions and real-world scenarios described in use cases.

Purpose:

To validate the software from an end-user perspective, ensuring workflows function as intended.

How to Perform:

- Identify use cases from requirements or user stories.
- Map steps and interactions for each use case.
- Test both positive and negative flows.

Example:

Testing an e-commerce checkout process:

- Positive Flow: Add items → Proceed to checkout → Enter payment → Place order
- Negative Flow: Add items → Proceed to checkout → Insufficient balance → Payment fails

When to Use:

When user experience and workflows are critical.

Advantages:

- Mimics real-world usage scenarios.
- Ensures coverage of end-to-end workflows.

Limitations:

- May miss low-level functional issues.

F) Random Testing

Inputs are generated randomly to test the system's robustness and error handling.

Purpose:

To uncover hidden issues or vulnerabilities by testing with unpredictable inputs.

How to Perform:

- Use random data generators or create diverse test data manually.
- Execute tests with randomly generated inputs.

Example:
Entering random strings, numbers, or special characters into a text field.
When to Use:
For exploratory or stress testing scenarios.
Advantages:

- Can uncover unexpected defects.
- Useful for negative testing.

Limitations:

- Lacks structured coverage.
- Difficult to reproduce defects.

G) Graph-Based Testing

Uses graphs to represent objects (nodes) and their relationships (edges).
Purpose:
To validate complex systems with interdependent components or navigation flows.
How to Perform:

- Create a graph representing system components and relationships.
- Design test cases for all possible paths or interactions.

Example:
Testing navigation paths in a website or app.
When to Use:
For systems with navigation flows, dependencies, or data relationships.
Advantages:

- Provides visual clarity for complex systems.
- Ensures all paths are covered.

Limitations:

- May become cumbersome for highly interconnected systems.

5.3 White Box Testing Techniques

White box testing, also known as **clear-box testing**, **glass-box testing**, or **structural testing**, is a software testing method that examines the internal structure, design, and code of the application. Unlike black box testing, white box testing requires knowledge of the code and focuses on verifying the internal workings of the application. White box testing involves testing the application's internal logic and structure. Testers design test cases based on the code structure to ensure all possible paths, branches, and statements are executed at least once. White box testing relies on a variety of techniques to achieve thorough code coverage. These techniques ensure that every part of the program is executed and validated. Below are the key types of white box testing techniques explained in detail.

A) Key Characteristics of White Box Testing

- **Code Visibility:** Testers need access to the source code.
- **Focus on Internal Logic:** Tests verify whether the code operates as intended.
- **Test Coverage Metrics:** Metrics like statement, branch, and path coverage are used to assess completeness.
- **Automated Tools:** Often supported by tools for efficient execution.

B) When to Perform White Box Testing

- **During Unit Testing:** Developers use it to test individual code components.
- **During Integration Testing:** To test interactions between integrated modules.
- **When Code Changes Occur:** To ensure new changes do not introduce defects.

C) Tools for White Box Testing

- **JUnit**: For unit testing Java applications.
- **NUnit**: For testing .NET applications.
- **CppUnit**: For C++ applications.
- **SonarQube**: For code analysis and quality checks.
- **EclEmma**: For code coverage analysis in Java.
- **Karma and Jasmine**: For JavaScript testing.

D) Advantages

- **Thorough Coverage:** Detects internal logic errors.
- **Early Bug Detection:** Identifies defects during development.
- **Improves Code Quality:** Encourages cleaner, more testable code.

E) Disadvantages

- **Requires Code Knowledge:** Testers need programming expertise.
- **Time-Consuming:** Designing tests for complex logic takes time.
- **Not User-Focused:** Lacks focus on user experience.

F) Best Practices

- **Automate Repetitive Tests:** Use tools for efficiency.
- **Focus on Critical Paths:** Prioritize testing of high-risk areas.
- **Combine with Black Box Testing:** Ensure comprehensive coverage.

5.4 Types of White Box Testing Techniques

White box testing relies on a variety of techniques to achieve thorough code coverage. These techniques ensure that every part of the program is executed and validated. Below are the key types of white box testing techniques explained in detail.

A) Statement Coverage

Statement Coverage ensures that every line of code in the application is executed at least once during testing.

Purpose:

To verify that no part of the code is left untested.

How to Perform:

1. Analyze the source code and identify all executable statements.
2. Write test cases to ensure each statement is executed.

Example:

java

Copy code

```
if (x > y) {
   max = x;
} else {
   max = y;
}
```

- Test Case 1: x = 10, y = 5 → Executes max = x;
- Test Case 2: x = 3, y = 7 → Executes max = y;

When to Use:

- During unit testing to ensure basic execution of code blocks.

Advantages:

- Simple to implement and understand.

- Ensures all parts of the code are at least touched.

Limitations:

- Does not test conditional outcomes.
- May miss complex errors related to logic.

B) Branch Coverage

Branch Coverage, also known as **Decision Coverage**, ensures that every branch (true/false) of decision points like if, else, switch statements is tested.

Purpose:

To test all possible outcomes of conditional logic.

How to Perform:

1. Identify all decision points in the code.
2. Create test cases that result in both true and false outcomes for each decision.

Example:

For the following code:

java

Copy code

```
if (a > b) {
   max = a;
} else {
   max = b;
}
```

- Test Case 1: a = 5, b = 3 → Tests a > b as true.
- Test Case 2: a = 3, b = 5 → Tests a > b as false.

When to Use:

- For applications with multiple decision points or conditions.

Advantages:

- Detects logical errors and ensures robust code.
- Provides higher coverage than statement testing.

Limitations:

- Does not test the flow through all combinations of conditions.

C) Path Coverage

Path Coverage tests all possible execution paths in the program, including all combinations of decisions and loops.

Purpose:

To verify every independent path is executed at least once.

How to Perform:

1. Create a control flow graph for the code.
2. Identify all possible paths through the code.
3. Write test cases to cover each path.

Example:

For code with nested conditions:

java

Copy code

```
if (x > 0) {
   if (y > 0) {
      result = x + y;
   }
}
```

Paths:

x > 0 and y > 0
x > 0 and y <= 0
x <= 0

When to Use:

- When exhaustive testing of logic is required.

Advantages:

- Ensures complete coverage of all paths.

Limitations:

- Becomes impractical for programs with a large number of paths.

III) Condition Coverage

Condition Coverage tests each boolean sub-expression in decision points for true and false outcomes.

Purpose:

To validate all possible conditions independently.

How to Perform:

1. Identify all conditional expressions.
2. Create test cases to cover each condition's true and false outcomes.

Example:

For the condition if (x > 0 && y > 0):

- Test Case 1: x = 5, y = 5 → Both true.
- Test Case 2: x = 5, y = -5 → One true, one false.
- Test Case 3: x = -5, y = 5 → One false, one true.
- Test Case 4: x = -5, y = -5 → Both false.

When to Use:

- For applications with complex conditional logic.

Advantages:

- Identifies logic errors in individual conditions.

Limitations:

- Does not test interactions between conditions.

D) Loop Testing

Loop Testing focuses on validating the behavior of loops in the code.

Purpose:

To ensure loops execute correctly under various conditions.

Types of Loop Testing:

1. **Simple Loops:** Test for zero iterations, one iteration, and multiple iterations.
2. **Nested Loops:** Test inner and outer loops independently and in combination.
3. **Concatenated Loops:** Test sequences of independent loops.

How to Perform:

- Identify all loops in the program.
- Test loop boundaries, including:
  - No iterations (skipping the loop).
  - Exactly one iteration.
  - Multiple iterations.
  - Maximum iterations (if applicable).

Example:

For a loop that processes an array:

java

Copy code

```
for (int i = 0; i < n; i++) {
   process(array[i]);}
```

- Case 1: n = 0 (No iterations).
- Case 2: n = 1 (Single iteration).
- Case 3: n = 5 (Multiple iterations).

Advantages:

- Ensures loops handle edge cases.

Limitations:

- Complex for deeply nested or interdependent loops.

E) Control Flow Testing

Control Flow Testing examines the program's flow of control using control flow graphs.

Purpose:

To identify issues in sequences of control statements.

How to Perform:

1. Represent the program's control flow using a graph.
2. Identify all nodes (statements) and edges (branches).
3. Design test cases to traverse all edges.

Advantages:

- Provides a visual representation of program flow.

Limitations:

- Not suitable for non-sequential logic.

F) Data Flow Testing

Data Flow Testing checks the use and flow of variables in the program.

Purpose:

To detect issues like uninitialized variables, redundant code, or incorrect data usage.

How to Perform:

1. Identify all variables and their lifecycle (definition, usage, deletion).
2. Validate the correct flow of data through test cases.

Advantages:

- Identifies defects related to data handling.

Limitations:

- Can become complex for large codebases.

G) Mutation Testing

Mutation Testing introduces small changes (mutations) to the code and checks if the test suite can detect them.

Purpose:

To validate the effectiveness of test cases.

How to Perform:

1. Modify code slightly (e.g., changing operators, values).
2. Execute test cases to ensure they fail due to the mutation.

Advantages:

- Ensures the robustness of test cases.

Limitations:

- Computationally expensive.

5.5 Experience-Based Testing Techniques

Experience-based testing techniques leverage the skills, intuition, domain knowledge, and prior experience of testers to identify potential issues in software. These techniques are especially useful when detailed requirements or specifications are unavailable, or when testing needs to focus on real-world usage scenarios. Experience-based testing relies on the tester's expertise, judgment, and past encounters with similar applications or technologies. It is less structured than formal techniques but equally effective, especially in exploratory or fast-paced testing environments.

A) Key Objectives

- To uncover defects that are not easily identified by scripted or automated testing methods.
- To simulate real-world user behavior.
- To provide quick feedback in dynamic or agile environments.

B) When to Use Experience-Based Testing?

- When detailed requirements or specifications are missing or incomplete.
- During exploratory testing phases where creativity and intuition play a role.
- To supplement formal testing techniques.
- When time or resources are limited, and rapid defect identification is crucial.

C) Advantages of Experience-Based Testing

- Encourages creativity and adaptability in testing.
- Allows testers to focus on high-risk areas based on intuition and past knowledge.
- Complements structured techniques by covering gaps or blind spots.
- Provides flexibility in test execution.

5.6 Types of Experience-Based Testing Techniques

Experience-based techniques encompass a variety of approaches, each tailored to specific scenarios. Below is a detailed explanation of the major types:

A) Exploratory Testing

Exploratory Testing is an unscripted approach where testers simultaneously learn about the application, design test cases, and execute tests.

Purpose:

- To discover defects in areas that might be overlooked by formal testing.
- To uncover usability, compatibility, or real-world issues.

How to Perform:

1. **Charter Creation:** Define a scope or goal for the testing session (e.g., "Test the login functionality").
2. **Explore:** Interact with the application dynamically, observing behaviors and identifying issues.
3. **Note Observations:** Record test cases, defects, and areas of interest for future exploration.

Example Scenario:

Testing an e-commerce website’s checkout process by:

- Trying different combinations of payment methods.
- Testing with invalid coupon codes.
- Checking how the system handles session timeouts.

Tools to Assist:

- Session-based Test Management (SBTM).
- Tools like Testpad or exploratory add-ons in tools like JIRA.

Advantages:

- Encourages creativity and adaptability.

- Allows for quick identification of high-impact issues.

Disadvantages:

- Lacks repeatability unless observations are well-documented.
- Relies heavily on the tester's expertise.

B) Error Guessing

Error Guessing involves predicting potential problem areas in the application based on the tester's experience, domain knowledge, and understanding of common error patterns.

Purpose:

- To target known areas of vulnerability or past issues.
- To identify defects based on common errors.

How to Perform:

1. **Leverage Knowledge:** Use historical defect data, similar applications, or known issues in the domain.
2. **Formulate Tests:** Write test cases to target these error-prone areas.
3. **Execute Tests:** Validate if similar defects exist.

Example Scenario:

In a financial application, testers might focus on areas prone to numerical precision issues or boundary conditions (e.g., maximum transaction limits).

When to Use:

- When historical defect patterns are known.
- To supplement other techniques during high-risk testing.

Advantages:

- Quick identification of known defect patterns.
- Effective in high-risk or critical modules.

Disadvantages:

- Highly dependent on the tester's knowledge and intuition.
- May miss defects outside known patterns.

C) Checklist-Based Testing

Checklist-Based Testing involves creating a structured list of items to verify based on prior experience or domain knowledge.

Purpose:

- To ensure coverage of critical areas without relying on detailed documentation.
- To provide a structured approach in an unstructured testing environment.

How to Perform:

1. **Create a Checklist:** Develop a list of test items based on knowledge, previous projects, or critical business functions.
2. **Execute Tests:** Work through the checklist systematically.
3. **Update Continuously:** Add new items based on observations during testing.

Example Checklist:

For testing a login system:

- Verify successful login with valid credentials.
- Test login with invalid credentials.
- Check session expiration behavior.

Tools:

- Spreadsheet tools (Excel, Google Sheets).
- Test management tools like TestRail or Zephyr.

Advantages:

- Provides a quick reference for critical tests.

- Ensures systematic coverage of key areas.

Disadvantages:

- Can be less effective for discovering new defects.
- Checklist quality depends on prior knowledge.

D) Ad-Hoc Testing

Ad-Hoc Testing is an informal and unstructured testing approach where testers examine the application based on their instincts and experience.

Purpose:

To find defects quickly without the constraints of predefined test cases.

How to Perform:

1. No predefined plans; start testing based on intuition.
2. Focus on high-risk or complex areas.
3. Document findings for further exploration.

Example Scenario:

Testing an image upload feature by:

- Uploading unsupported file types.
- Testing file size limits.
- Interrupting the upload process midway.

Advantages:

- Highly flexible and fast.
- Useful for identifying random defects.

Disadvantages:

- Lacks repeatability and documentation.
- Relies on the tester's expertise.

E) Fault Attack Testing

Fault Attack Testing involves targeting specific known weaknesses or vulnerabilities in the application based on experience or historical defect data.

Purpose:

- To identify defects based on known fault patterns.

How to Perform:

1. Use historical defect data to identify common vulnerabilities.
2. Focus test cases on these fault patterns.

Example:

- Testing an application for SQL injection vulnerabilities by intentionally crafting invalid queries.

Advantages:

- Highly effective in security and vulnerability testing.

Disadvantages:

- Limited to known fault patterns.

5.7 Grey Box Testing: A Key Testing Technique

Grey Box Testing is a hybrid testing method that combines aspects of both **black box** and **white box testing**. It involves having partial knowledge of the internal workings of the application while still focusing on functional testing, much like in black box testing. This gives testers a unique perspective, allowing them to design more effective test cases while still not being fully immersed in the source code.

In **Grey Box Testing**, the tester has **limited knowledge of the internal workings** of the system or application, such as the architecture, database design, or APIs. This contrasts with **black box**

testing, where the tester has no access to internal information, and **white box testing**, where the tester has complete access to the source code.
This middle ground allows testers to design test cases that are **more focused** and **better informed**, while still ensuring that they simulate real-world user behavior, without delving too deeply into the code itself.

A) When to Use Grey Box Testing?

- **Integration and system testing** where having an understanding of how modules work together can help testers identify flaws that wouldn't be visible in black box testing alone.
- **Testing complex applications** that have both visible user-facing components and intricate internal logic (like web services or APIs).
- **When performing security or penetration testing**, where knowledge of the internal structure is beneficial for identifying vulnerabilities, but the complete source code isn't necessary.

B) How to Perform Grey Box Testing

- **Analyze the Application:** Get a high-level understanding of the system's architecture, design, and data flow, typically provided by the development team.
- **Design Test Cases:** Based on partial knowledge of the application's internal structure, design test cases that cover both functional and non-functional requirements. This might include focusing on integration points, potential vulnerabilities, or user interface behavior that may depend on backend systems.
- **Execute and Monitor:** Run the designed tests and analyze how the system behaves from both a user perspective (black box) and an internal perspective (white box). Ensure to test interactions that involve internal logic and external components.
- **Track Results and Report:** Document defects based on findings, which could relate to either the external user interface or hidden vulnerabilities and integration issues in the backend system.

C) Advantages of Grey Box Testing

- **Informed Test Design:** Grey box testing allows for more informed testing than black box, as testers can design better test cases by understanding the internal structure of the application, even if they do not have full access to the code.
- **Faster than White Box Testing:** It's more efficient than white box testing because testers do not need to access or understand the full code base.
- **Unbiased Testing:** Unlike white box testing, testers are not limited by the developers' code or logic, leading to better results from a user perspective.
- **Effective for Complex Systems:** It is ideal for testing systems that have both a user-facing component and an intricate backend system, as it ensures thorough coverage across both levels.

D) Disadvantages of Grey Box Testing

- **Limited Knowledge:** Since testers have partial access to internal information, there could still be gaps in the testing, leading to missed issues that might be uncovered with full access.
- **Complex Test Design:** Combining internal knowledge with functional tests can be challenging, as testers must balance both the user interface and system architecture considerations.
- **Requires Skilled Testers:** Testers need a blend of skills in both user-centric functional testing and technical understanding of system internals, which may require more experienced personnel.

6. When & Why to Use Static and Dynamic Techniques

Technique	When to Use	Why to Use
Static Reviews	Early stages (requirements/design).	To prevent defects in initial phases.
Static Analysis	During coding.	To detect vulnerabilities and code smells.
Black Box Techniques	Functional testing phases.	To validate user-facing functionalities.
White Box Techniques	Unit and integration testing.	To ensure code logic is correct.
Experience-Based Testing	Minimal documentation or time.	To explore defect-prone areas dynamically.

7. Tools Supporting Static and Dynamic Techniques

Technique	Tools
Static Code Analysis	SonarQube, Coverity, Checkstyle
Black Box Testing	Selenium, Postman, JMeter
White Box Testing	JUnit, NUnit, TestNG
Experience-Based Testing	TestRail, QTest, Jira

8. Best Practices for Applying Testing Techniques

1. Combine multiple techniques for comprehensive coverage.
2. Apply static techniques early to prevent defects.
3. Use dynamic techniques iteratively to detect defects during execution.
4. Leverage automation for repetitive tasks.
5. Train teams in diverse techniques to maximize effectiveness.

9. Common Pitfalls to Avoid

1. Over-reliance on a single technique.
2. Skipping static testing, leading to defects in requirements or design.
3. Neglecting exploratory testing for unexpected scenarios.
4. Ignoring test coverage metrics during white box testing.