Software Testing
Assignment 2

Q1. Explain the significance of boundary conditions in black-box testing and describe how they impact test case design.

Significance:
Boundary conditions refer to the values at the **edge of input ranges** (e.g., minimum, maximum, just inside, just outside). Most software defects occur at these boundaries because developers often make mistakes in handling extreme values.

Impact on Test Case Design:

- Testers design cases using **Boundary Value Analysis (BVA)**.
- Instead of testing all values in a range, they test:
  - **Minimum value**
  - **Minimum + 1**
  - **Maximum – 1**
  - **Maximum value**
  - **Values just outside boundary**

Example:
If valid age is 18–60, test values: 17, 18, 19, 59, 60, 61.

This leads to:

- Better defect detection
- Reduced number of test cases
- Higher reliability in input validation

Q2. Differentiate between static and structural approaches in white-box testing, with examples.

Static Approach

- Involves examining code **without executing it**.
- Techniques:
    - Code walkthroughs
    - Code inspections
    - Static analysis tools
- Useful when:
    - Detecting syntax errors, logical flaws
    - Enforcing coding standards early in SDLC

Structural Approach

- Tests the **internal structure** by executing the code.
- Techniques:
    - Statement coverage
    - Branch coverage
    - Path coverage
- Useful when:
    - Ensuring all code paths execute at least once
    - Identifying runtime issues

Q3. What is mutation testing in advanced white-box testing, and how does it assess test effectiveness?

Mutation testing introduces **small changes (mutations)** in the source code, such as:

- Changing `>` to `<`
- Replacing `+` with `-`
- Removing a statement

These changed versions are called **mutants**.

How it assesses test cases:

- Test cases are executed on each mutant.
- If a test case fails and detects the mutation → **mutant is killed**.
- If the test case does NOT detect the mutation → **mutant survives**, revealing weak test cases.

Benefit:

Helps measure how well test cases detect real faults → improves test quality.

Q4. Describe the key methodologies used in test case design. How do black-box and white-box testing contribute?

Key Test Case Design Methodologies

1. **Equivalence Partitioning**
2. **Boundary Value Analysis**
3. **Decision Table Testing**
4. **State Transition Testing**
5. **Cause–Effect Graphing**
6. **Control Flow–based Testing**

7. **Data Flow Testing**

Contribution of Testing Techniques

Black-Box Testing

- Based on **requirements and functionality**.
- Ensures:
  - Correct output for valid and invalid inputs
  - User-focused validation
  - Requirement coverage

White-Box Testing

- Based on **internal code structure**.
- Ensures:
  - Maximum coverage of statements, branches, and paths
  - Logical correctness
  - Detection of unreachable code

Together they create **comprehensive functional + structural coverage**.

Q5. Explain control flow in white-box testing and how visualizing it improves test coverage.

Control Flow:
Represents the **order in which statements or instructions execute** in a program.

Tools:

- Control Flow Graph (CFG)
  - Nodes → statements/blocks

    - Edges → flow of execution

Benefits of Visualizing Control Flow:

- Identifies all possible execution paths
- Helps design:
    - Statement coverage tests
    - Branch coverage tests
    - Path coverage tests
- Detects:
    - Unreachable code
    - Loops and complex structures
- Improves test thoroughness and ensures coverage of edge paths

---

Q6. Importance of aligning test cases with requirements (Requirements-Based Testing).

Why it is important:

- Requirements define **what the system must do**.
- Test cases aligned to requirements ensure:
    - No functionality is missed
    - All user expectations are validated
    - Early detection of requirement defects

How Requirements-Based Testing Helps:

- Creates **traceability** between requirements and test cases
- Ensures **full coverage**

- Helps identify:
    - Missing requirements
    - Ambiguous requirements
- Reduces defect leaks to later stages

Q7. Compare and contrast black-box and white-box testing.

Aspect	Black-Box Testing	White-Box Testing
Focus	Functional behavior	Internal structure
Knowledge Required	No code knowledge	Full code knowledge
Test Basis	Requirements	Code and logic
Techniques	BVA, EP, Decision Tables	Statement/Branch/Path coverage
Advantages	User-centric, detects missing requirements	High coverage of code paths
Limitations	Cannot detect hidden code bugs	Time-consuming, needs skilled testers
Best Use	Acceptance and system testing	Unit testing, security testing

Conclusion: Both complement each other in achieving reliable software.

Q8. Explain the role of test adequacy metrics.

Test Adequacy Metrics:
Measure how complete or effective the testing is.

Common Adequacy Criteria:

1. **Statement Coverage**
   % of executed statements.

2. **Branch Coverage**
   % of executed decision outcomes (true/false).

3. **Path Coverage**
   % of independent execution paths covered.

4. **Condition Coverage**
   Tests each boolean condition.

Impact on Software Quality

- Ensures thorough testing
- Identifies untested parts of software
- Improves reliability
- Helps in measuring test effectiveness objectively

Q9. Describe various white-box testing techniques (statement, branch, path coverage).

1. Statement Coverage

Ensures every statement is executed at least once.

Purpose:
Detects missing or unused statements.

---

2. Branch Coverage

Ensures every branch (true/false) of decision points is executed.

Purpose:
Catches logical errors missed by statement coverage.

---

3. Path Coverage

Ensures every possible execution path is tested.

Purpose:
Most thorough; covers combinations of branches.

Challenges:
Number of paths grows exponentially; not always practical.

Q10. Discuss mutation testing in detail. How does it improve test quality and what are its challenges?

Detailed Explanation:

Mutation testing intentionally introduces small faults into the program. Mutants simulate common developer mistakes.

Types of Mutations:

- Arithmetic operator changes
- Logical operator changes
- Constant replacement
- Variable replacement
- Statement removal

Improvement in Test Quality:

- Detects weak or ineffective test cases
- Encourages writing stronger test scenarios
- Measures test suite strength
- Validates correctness of existing test cases

Challenges:

1. **High Computational Cost**
   Many mutants → heavy execution time.

2. **Equivalent Mutants**
   Mutants that behave exactly like original code and are impossible to kill.

3. **Complexity**
   Difficult to apply for large systems.

4. **Automation Required**
   Requires specialized tools (e.g., PIT, MuJava).