

# ECE 322 SOFTWARE TESTING AND MAINTENANCE

## COMPREHENSIVE SHORT-ANSWER EXAM SOLUTIONS

### SECTION 1: FUNDAMENTALS (Modules 1-3)

#### Q1. Define the difference between an error, a fault, and a failure.

**Answer:** Error is a human action that produces an incorrect result (e.g., programmer types wrong variable name). Fault is an incorrect step, process, or data definition in code (e.g., bug in the program). Failure is the inability of the system to perform required functions within specifications (e.g., system crashes). Relationship: Errors → Faults → Failures.

#### Q2. Distinguish between verification and validation.

**Answer:** Verification concerns the process (building product right) - focuses on checking conformance to internal specifications. Validation concerns the product (building the right product) - ensures software meets customer requirements and quality expectations. Verification asks 'Are we building the product right?' while validation asks 'Are we building the right product?'

#### Q3. Explain coincidental correctness.

**Answer:** Coincidental correctness occurs when a test point follows an incorrect execution path, but the output variables coincidentally match the expected correct output. This is problematic because it masks faults - the test passes despite the code having bugs, giving false confidence in code quality.

#### Q4. What is test coverage? Name three types.

**Answer:** Test coverage measures the degree to which source code has been executed during testing. Three types: (1) Statement coverage - percentage of statements executed, (2) Branch/decision coverage - percentage of branches taken, (3) Path coverage - percentage of execution paths traversed.

#### Q5. State three fundamental principles of software testing.

**Answer:** (1) Testing can show presence of faults, never their absence (Dijkstra's principle), (2) Exhaustive testing is impossible for non-trivial programs, (3) Early testing reduces cost of fixing defects, (4) Defects cluster in certain modules, (5) Tests wear out - repeated tests find fewer new bugs.

### SECTION 2: BLACK-BOX TESTING (Module 5)

#### Q6. Define equivalence class partitioning.

**Answer:** Equivalence class partitioning divides the input domain into classes where all values in the same class are expected to behave similarly. Assumption: testing one representative value from each equivalence class is equivalent to testing any other value in that class. This reduces test cases while maintaining reasonable coverage.

#### Q7. Distinguish between weak normal and strong normal equivalence class testing.

**Answer:** Weak normal uses the single fault assumption - design test cases covering one valid value from each equivalence class and as many valid classes as possible simultaneously. Strong normal uses the multiple fault assumption - select test cases from the Cartesian product of equivalence classes, testing all possible combinations of inputs.

#### Q8. What is boundary value analysis?

**Answer:** Boundary value analysis tests values at the edges of equivalence classes (minimum, just above minimum, maximum, just below maximum). Bugs cluster at boundaries due to off-by-one errors, incorrect comparison operators ( $<$  vs  $\leq$ ), and edge case handling failures.

#### Q9. Explain decision tables in black-box testing.

**Answer:** Decision tables systematically document input conditions (causes) and corresponding actions (effects). They map all combinations of conditions to expected outcomes. Decision table reduction eliminates redundant or impossible combinations using constraints, reducing the number of required test cases.

#### Q10. Define operational profile and when NOT to use it.

**Answer:** Operational profile is a list of disjoint operations and their probabilities of occurrence, allowing testing to mirror actual usage patterns. Do NOT use when: (1) Software is small-scale, (2) Few users or homogeneous user groups, (3) Non-diverse usage environment, (4) Lack of detailed statistical usage data.

#### Q11. Relationship between Petri nets and finite state machines.

**Answer:** A finite state machine is a special case of a Petri net. When each transition of the Petri net has a single input place AND a single output place, the Petri net becomes a finite state machine.

#### Q12. Extreme point combination testing.

**Answer:** Extreme points are the boundary values (minimum and maximum) of each input variable's domain. Extreme point combination testing tests all combinations of extreme values to catch boundary interaction faults. For  $n$  variables with 2 extreme points each, this generates  $2^n$  test cases.

### SECTION 3: WHITE-BOX TESTING (Module 6)

#### Q13. Draw CFG and calculate cyclomatic complexity.

**Answer:** CFG has 2 nested if statements. Cyclomatic complexity = 3 (using formula: # of decision nodes + 1 = 2 + 1 = 3). Can also calculate as:  $e - n + 2$  where  $e$ =edges,  $n$ =nodes, or count planar regions.

#### Q14. Define statement, branch, and path coverage. Rank them.

**Answer:** Statement coverage: percentage of statements executed. Branch coverage: percentage of branches (true/false) executed. Path coverage: percentage of all possible execution paths tested. Ranking from weakest to strongest: Statement < Branch < Path. Path coverage subsumes branch coverage, which subsumes statement coverage.

#### Q15. McCabe's cyclomatic complexity calculation methods.

**Answer:** Cyclomatic complexity  $v(G)$  measures code complexity. Three calculation methods: (1)  $v(G) = e - n + 2$  (edges - nodes + 2), (2)  $v(G)$  = number of binary decision nodes + 1, (3)  $v(G)$  = number of enclosed planar regions in the CFG.

#### Q16. Data flow testing anomalies.

**Answer:** dd (define-define): variable defined twice without use in between - wastes computation. du (define-undefine): variable defined then destroyed without use - logic error. ur (undefine-reference): variable used before definition - potential uninitialized variable error.

### SECTION 4: ADVANCED TESTING (Module 7)

#### Q17. Differentiate fault injection and fault seeding.

**Answer:** Fault injection: deliberately introducing faults during testing to evaluate system robustness and error handling. Fault seeding (Mills): inserting known faults into code to estimate total number of faults. Formula: Confidence  $C = S / (S + N + 1)$  if  $n \leq N$ , where  $S$ =seeded faults,  $N$ =actual faults found,  $n$ =seeded faults discovered.

### Q18. Explain mutation testing and three mutation types.

**Answer:** Mutation testing creates modified versions (mutants) of the program by introducing small changes. Test quality is measured by mutation score (percentage of mutants killed). Three types: (1) Value mutations - change constants/variables, (2) Decision mutations - modify logical conditions, (3) Statement mutations - alter control flow statements.

### Q19. What is a testing oracle?

**Answer:** A testing oracle is a mechanism for determining expected outputs for test inputs. The oracle problem is fundamental: how do we know if test output is correct? Solutions include: specifications, previous versions, parallel implementations, metamorphic relations, or manual verification.

### Q20. 100% branch coverage and 80% mutation score - fault-free?

**Answer:** No. Testing can only show the presence of faults, never their absence (Dijkstra). High coverage doesn't guarantee fault-free software. 20% of mutants survived, indicating test suite has gaps. Also, coverage measures paths executed, not correctness of outputs.

## SECTION 5: INTEGRATION & REGRESSION (Module 8)

### Q21. Top-down vs bottom-up integration. When to choose top-down?

**Answer:** Top-down: integrate from main module downward using stubs. Bottom-up: integrate from lowest modules upward using drivers. Choose top-down in unstable environments because it avoids premature commitment to unstable interfaces - high-level design is tested first while low-level details can be refined later.

### Q22. Define driver and stub. Which strategy uses each?

**Answer:** Driver: simulator of a calling program (dummy program unit) that feeds test inputs to the module under test - used in bottom-up integration. Stub: simulator of a called program (dummy program unit) that returns predetermined responses - used in top-down integration testing.

### Q23. What is big-bang integration testing?

**Answer:** Big-bang (non-incremental) integration: all modules integrated simultaneously and tested as a single entity. Disadvantages: (1) Hard to isolate faults, (2) No systematic approach, (3) Interface errors difficult to diagnose, (4) Debugging is complex, (5) Late detection of integration issues.

### Q24. Explain regression testing.

**Answer:** Regression testing re-runs previously passing tests after code changes to ensure modifications didn't break existing functionality. Run when: code is modified, bugs are fixed, new features added, or environment changes. Detects unintended side effects of changes.

## SECTION 6: SOFTWARE MAINTENANCE (Module 9)

### Q25. Define S-type, P-type, and E-type systems.

**Answer:** S-type (Specifiable): problem formally defined with well-defined solution; correctness-focused (e.g., sorting algorithm). Software does not evolve. P-type (Problem-solving): solves a problem but specs may be incomplete; approximation acceptable (e.g., chess program). E-type (Embedded): embedded in real-world context; must adapt to changing environment (e.g., payroll system). E-type systems continuously evolve.

### Q26. What is software maintainability?

**Answer:** Software maintainability is the ease with which software can be modified to fix faults, improve performance, or adapt to changed requirements. Factors affecting it: (1) Code complexity (cyclomatic/essential complexity), (2) Documentation quality, (3) Code modularity, (4) Coding standards adherence.

### Q27. Define four categories of software maintenance.

**Answer:** (1) Corrective: fixing bugs/faults discovered during operation. (2) Adaptive: modifying software for new platforms or environments. (3) Perfective: enhancing performance or improving maintainability. (4) Preventive: making changes to prevent future problems or improve reliability.

### Q28. System re-engineering vs reverse engineering.

**Answer:** Reverse engineering: analyzing existing system to identify components and relationships, extracting design information from implementation. Re-engineering: restructuring/rewriting existing system to improve quality and maintainability while preserving functionality. Re-engineering often includes reverse engineering as the first step.

### Q29. Why is essential complexity $ev(G)$ more critical than $c(G)$ ?

**Answer:** Cyclomatic complexity  $c(G)$  increases gradually as code grows, showing continuity. Essential complexity  $ev(G)$  measures unstructured control flow and can increase abruptly when structural changes are made during maintenance. Sudden  $ev(G)$  spikes indicate deteriorating code structure requiring refactoring.

## SECTION 7: SOFTWARE RELIABILITY (Module 11)

### Q30. How does software reliability differ from hardware reliability?

**Answer:** Hardware reliability: failures due to physical wear, aging, environmental factors - follows bathtub curve. Software reliability: failures due to design faults in code - no physical wear-out. Software doesn't degrade over time; faults exist from development. Software failures are deterministic (same inputs  $\rightarrow$  same failure).

### Q31. Name time-dependent and time-independent reliability models.

**Answer:** Time-dependent: model failure rate as function of time (e.g., Jelinski-Moranda model, Musa execution time model). Time-independent: model reliability based on other factors like fault count or test cases (e.g., Mills fault seeding model, input domain-based models).

### Q32. High-quality but low-reliability software?

**Answer:** Yes. Quality is multi-faceted (usability, maintainability, efficiency, etc.). A system could have excellent documentation, clean code, good UI but still crash frequently or produce incorrect results. Example: beautifully-designed web retailer with frequent database connection failures. However, if ISO standards are required, reliability becomes mandatory.

## SECTION 8: SCENARIO-BASED QUESTIONS

### Q33. Triangle classification module (a,b,c).

**Answer:** (a) Input domain: 3-dimensional space of positive real numbers ( $a, b, c > 0$ ). (b) Valid classes: equilateral ( $a=b=c$ ), isosceles (2 sides equal), scalene (all different). Invalid classes: non-positive values, triangle inequality violations ( $a+b \leq c$ ,  $a+c \leq b$ ,  $b+c \leq a$ ), non-numeric inputs. (c) Boundary values: test minimums (0.001, 1), maximums (large values), equality boundaries ( $a=b$ ,  $b=c$ ,  $a=c$ ), and triangle inequality edges ( $a+b=c+\epsilon$ ).

### Q34. Linear system solver $Ax=b$ .

**Answer:** (a) Valid equivalence classes:  $\{A \mid \det(A) \neq 0\}$  (unique solution exists),  $\{A \mid \det(A) = 0\}$  (no unique solution). (b) For overdetermined system ( $m$  equations,  $n$  unknowns,  $m > n$ ): Use least squares solution  $x = (A^T A)^{-1} A^T b$ . Valid classes:  $\{A \mid \det(A^T A) \neq 0\}$ ,  $\{A \mid \det(A^T A) = 0\}$ .

### Q35. Web registration form with cause-effect graphing.

**Answer:** Causes: C1-name filled, C2-name valid format (letters/spaces only), C3-address filled, C4-zip filled, C5-city filled, C6-course# filled, C7-course# exists in system. Effects: E1-registration successful, E2-error message. Constraints reduce decision table from  $2^7=128$  to practical subset. Valid registration requires ALL causes true.

### Q36. Configuration testing: combinatorial explosion.

**Answer:** Total test cases =  $2 \times 3^6 = 2 \times 729 = 1,458$  test cases for exhaustive combinatorial testing of all configurations. This demonstrates why combinatorial testing techniques (pairwise, n-way coverage) are necessary for large configuration spaces.

## SECTION 9: CRITICAL THINKING

### Q37. Why testing cannot prove absence of faults.

**Answer:** Dijkstra: 'Testing shows the presence of bugs, never their absence.' Exhaustive testing is impossible for non-trivial programs (infinite/astronomical input space). Testing samples the input domain; passing tests only show those specific cases work, not that all cases work. A fault may exist in an untested execution path.

### Q38. Metamorphic tests pass - fault-free guarantee?

**Answer:** No. Metamorphic testing verifies relationships between inputs/outputs (e.g.,  $f(2x) = 2f(x)$ ) but doesn't guarantee correctness of individual outputs. Tests may miss faults outside the tested metamorphic relations. Testing never guarantees fault-free software.

### Q39. Relationship between control flow and data flow testing.

**Answer:** Control flow testing focuses on execution paths and coverage criteria (statement/branch/path). Data flow testing focuses on variable lifecycle (define-use patterns). They're complementary: control flow can miss data anomalies (dd, du, ur); data flow can miss control logic faults. Together they provide more comprehensive testing.

### Q40. How do production rules guide syntax-driven testing?

**Answer:** Production rules define grammar for valid inputs (e.g., BNF for compiler testing). Each production rule generates test cases: terminal productions test actual values, non-terminal productions test structural variations. Coverage criteria ensure all productions/derivations are tested, systematically exploring the syntactic space.

---

END OF ANSWER KEY

Note: Brief answers provided as per exam instructions. Justifications included where required.