PART A (2 Marks)

Topic	Appeared In	Times
Interface Testing (define)	Dec21, Dec23, Dec24, May21, May24	5
State Testing (define)	Dec22, Dec24, May22, May23, May24	5
Debugging (define/purpose)	Dec21, Dec22, Dec24, May22, May23	5
Decision Tables (define/use/components)	Dec21, Dec22, Dec23, May21, May22, May24	6
Transition Testing	Dec21, Dec24, May21, May23, May24	5
Path Testing (define)	Dec23, Dec24, May22, May24	4
Syntax Testing (define/objectives)	Dec23, Dec24, May22, May24	4
State Graph (define)	Dec23, May21, May22, May24	4
Path Expression (define)	Dec22, May22, May23 (x2)	4
Goal/Purpose of Software Testing	Dec23, May21, May22, May24	4
Bug Prevention	Dec22, May21, May24	3
Bug (meaning/define)	Dec21, Dec24, May23	3
Software Quality (define)	Dec23, May22, May23	3
Linguistics Metrics	Dec22, May21, May23	3
Flow Graph / Flow Chart	Dec22, May21, May23	3
Data Flow Graph	Dec23, May22, May23	3
Types/Names of Bugs	May21, May24	2
Domain (define)	Dec21, May23	2
Metrics (define)	Dec21, Dec24	2
Transaction (define)	Dec22, May21	2
Stable State	May21, May24	2

PART B (5 Marks)

Topic	Appeared In	Times
Domain Testing (need/concept/model)	Dec21, Dec24 (x2), May21, May22, May24	6
State Table/Graph (notes/examples)	Dec22, May21, May22, May23, May24	5
Five Phases / Purpose of Testing	Dec21, Dec24, May21, May23, May24	5
Five Rules of Path Product	Dec21, Dec22, Dec23, Dec24	4
Classify Metrics	May21, May22, May23, May24	4
Importance of Bugs	Dec21, Dec22, Dec24	3
Structure/Structural Metric	Dec21, Dec23, Dec24	3
Transactional Flow Testing Techniques	Dec22, May21, May24	3
Control Flow Graph / Flow Graph elements	Dec22, May22 (x2)	3
Steps in Syntax Testing	Dec23, May21, May24	3

Topic	Appeared In	Times
Consequences/Types of Bugs	May22 (x2), May23	3
Principles of State Testing	Dec21, Dec24	2
Steps in System Testing	Dec22, May23	2
Interface Testing (explain)	May21, May23	2

PART C (10 Marks)

Topic	Appeared In	Times
Components of Decision Tables	Dec22, May21, May22, May24	4
Elaborate Model for Testing	Dec22, May21, May24	3
Transaction Flow Testing Techniques	Dec22, May21, May22	3
Data Flow Testing Strategies	Dec23, May22, May23	3
Linguistics Metrics (discuss)	Dec22, May21, May24	3
Testing vs Debugging (compare)	Dec21, Dec24	2
Concept of Path Testing	Dec21, Dec24	2
Steps of Syntax Testing	Dec21, Dec24	2
Logic-Based Testing with Examples	Dec21, Dec24	2
Domain Testing Model	May21, May23	2
State Graphs (detailed)	May22, May23	2
Domains and Testability/Interface	May23, May24	2
Three Distinct Kinds of Testing	Dec22 (Part B), May23 (Part C)	2

Software Testing — 2 Mark Answers

1. Interface Testing (Define)

Interface testing is the process of verifying the communication and interaction between two or more software components, modules, or systems (e.g., between a client and a server, or between two subsystems). It checks whether data is correctly passed, formatted, and interpreted across the interface boundaries.

Key points:

- Focuses on the "joints" where two components meet — parameters, data formats, timing, and error handling.
- Common interface bugs: incorrect parameter order, wrong data type, unit mismatches (e.g., inches vs. cm), timing/synchronization errors, and misunderstood protocols.
- It is especially important in systems with multiple modules developed independently, since each module may work correctly in isolation but fail when integrated.
- Interface testing is usually done during **integration testing**, after unit testing and before system testing.

- Example: Testing whether a login module correctly sends username/password data to the authentication server in the expected format.

---

2. State Testing (Define)

State testing is a testing technique in which the behavior of a system is checked based on its current **state** and how it transitions to other states in response to inputs/events. It is modeled using a **state graph** (a directed graph of states and transitions).

Key points:

- Useful for systems that behave differently depending on history — e.g., vending machines, protocol handlers, menu-driven software, telecom switches.
- A state is a condition in which the system's behavior is determined by past inputs.
- Test cases are designed to cover all states, all transitions, and ideally all paths between states.
- Principles: every state should be reachable, every transition should be tested with valid and invalid inputs, and unreachable/dead states or transitions must be identified as design errors.
- Example: Testing an ATM system — states like "Idle," "Card Inserted," "PIN Entered," "Transaction in Progress."

---

3. Debugging (Define / Purpose)

Debugging is the process of **locating, analyzing, and fixing** the root cause of a bug/defect once it has been detected during testing (or reported by a user). It follows testing — testing shows *that* a bug exists, debugging finds *why* and *fixes* it.

Purpose of Debugging:

- To identify the exact statement(s)/logic causing incorrect behavior.
- To correct the code without introducing new defects (regression).
- To restore the software to its expected, specified behavior.
- To improve reliability and correctness before release.
- To document the cause, so similar bugs can be prevented in future (root-cause analysis).

Debugging vs Testing: Testing is destructive (tries to break the system, done by testers); debugging is constructive (fixes the system, usually done by developers).

---

4. Decision Tables (Define / Use / Components)

A **decision table** is a tabular technique used to represent complex combinations of conditions and their corresponding actions in a clear, structured format. It is widely used to design test cases when a system's logic depends on multiple independent conditions.

Components:

1. **Condition Stub** – lists all the conditions/inputs.
2. **Condition Entries** – the possible values (True/False, Yes/No) for each condition, forming rules/columns.
3. **Action Stub** – lists all possible actions/outputs.
4. **Action Entries** – indicates which action(s) apply for each combination (rule).

Uses/Applications:

- Designing test cases for business rules with multiple conditions (e.g., loan approval, discount calculation).
- Ensuring all combinations of inputs are covered (no missing cases).
- Useful in requirement specification review to detect ambiguity or incompleteness.
- Used in payroll systems, insurance claim processing, and validation logic.

---

5. Transition Testing

Transition testing verifies the **transitions between states** in a state graph — i.e., it checks whether the system correctly moves from one state to another when a specific event/input occurs.

Key points:

- A transition is defined by: current state → input/event → next state → action/output.
- Test cases are designed to exercise each transition at least once (transition coverage).
- Includes testing **valid transitions** (expected inputs) and **invalid transitions** (unexpected inputs, to ensure the system handles them gracefully, e.g., by rejecting the input or raising an error).
- Helps uncover bugs like missing transitions, incorrect next-state assignment, or unreachable states.
- Example: In an order-processing system, testing the transition from "Order Placed" to "Order Shipped" only after "Payment Confirmed."

---

6. Path Testing (Define)

Path testing is a **white-box testing** technique in which test cases are designed to execute specific paths (sequences of statements/branches) through a program's **control flow graph (CFG)**, from entry to exit.

Key points:

- The program is represented as a flow graph of nodes (statements) and edges (control flow/branches).
- Goal: achieve maximum path coverage — ideally covering every independent path at least once (related to **Cyclomatic Complexity**, which gives the minimum number of independent paths).
- Helps detect logic errors, unreachable code, and infinite loops.
- More thorough than statement or branch coverage since it considers combinations of decisions.
- Example: For an `if-else` inside a loop, path testing ensures both branches are tested across different loop iterations.

---

7. Syntax Testing (Define / Objectives)

Syntax testing is a technique where test cases are generated based on the **formal grammar/syntax rules** (format) of the input data that a program accepts, to check whether the system correctly parses valid inputs and rejects invalid ones.

Objectives:

- To verify the system correctly accepts all valid input formats as per the defined grammar (BNF – Backus-Naur Form).
- To verify the system correctly rejects/handles invalid or malformed inputs without crashing.
- To test boundary and error-handling of command interpreters, compilers, protocol parsers, and data-entry fields.
- To automate generation of large numbers of test cases directly from the syntax rules (reduces manual effort).
- Example: Testing whether a compiler correctly flags a missing semicolon or bracket.

---

8. State Graph (Define)

A state graph (also called a state transition diagram) is a **directed graph** used to model the behavior of a system in terms of its **states**, the **transitions** between them, and the **inputs/events** that trigger each transition.

Key points:

- **Nodes/Circles** represent states of the system.
- **Directed edges (arrows)** represent transitions, labeled with the input/event that causes them and sometimes the output/action produced.
- Used to model event-driven systems: protocols, embedded systems, UI menu flows, vending machines.
- Forms the basis for designing state and transition test cases.
- Example: A traffic light system's state graph — Red → (timer expires) → Green → (timer expires) → Yellow → (timer expires) → Red.

9. Path Expression (Define)

A path expression is an **algebraic representation** of all the possible paths through a program's flow graph, expressed using a set of operators — typically **sequence (·), selection/union (+), and repetition (loop, denoted with a superscript like)* — over the edges/links of the graph.

Key points:

- Derived from the flow graph by systematically reducing nodes (node-by-node reduction / path algebra).
- Each link/edge is given a name; the path expression combines these using the operators to represent every possible route from start to end node.
- Useful in structured/path testing and in computing metrics like path count or path complexity.
- Basis for applying the "rules of path products" to simplify expressions (e.g., distributive, identity, loop rules).
- Example: A simple path expression for a straight-line program with one loop: `a · b* · c`.

10. Goal / Purpose of Software Testing

The primary goal of software testing is to **find defects/bugs** in software before it is delivered to the customer, and to verify that the software meets its specified requirements — ensuring quality, reliability, and correctness.

Purposes:

- To detect errors and defects as early as possible (reduces cost of fixing).
- To verify the software works according to requirements (Verification) and meets the user's actual needs (Validation).
- To ensure reliability, performance, security, and usability of the software.
- To build confidence in the software's correctness before release.
- To reduce the risk of failure in production and prevent financial/reputational loss.
- Note: Testing can show the *presence* of bugs, not their *absence* — it cannot guarantee a completely bug-free system.

11. Bug Prevention

Bug prevention refers to the set of practices and techniques used to **avoid the introduction of bugs** during the software development process, rather than finding and fixing them after they occur.

Key techniques:

- Following disciplined **software development processes/models** (e.g., structured design, code reviews, walkthroughs).
- Using **formal specifications and modeling techniques** to remove ambiguity in requirements.
- Applying **coding standards and static analysis tools** to catch errors early.
- **Test-first / early testing** approaches (e.g., designing test cases from requirements before coding).
- Learning from bug histories: analyzing recurring bug patterns to prevent similar mistakes.
- Bug prevention is more cost-effective than bug detection, since fixing defects late in development is significantly more expensive.

---

12. Bug (Meaning / Define)

A bug (also called a defect or fault) is a **flaw or error in a software program** that causes it to produce an incorrect, unexpected, or undesired result, or to behave in a way that differs from its intended/specified behavior.

Key points:

- Bugs can occur due to mistakes in requirements, design, coding, or documentation.
- A bug is only a bug in relation to a **specification** — if the specification is unclear/wrong, disagreement can arise about what qualifies as a bug.
- Bugs can be classified as functional, performance-related, UI-related, logic errors, boundary/domain errors, syntax errors, etc.
- The IEEE distinguishes: **Error** (human mistake) → **Fault/Defect** (flaw in code) → **Failure** (observable incorrect behavior caused by the fault).
- Example: A program that crashes when a negative number is entered has a boundary-handling bug.

---

13. Software Quality (Define)

Software quality refers to the **degree to which a software product meets specified requirements** (functional and non-functional) and satisfies user needs and expectations, while being free from defects.

Key attributes (quality factors):

- **Functionality** – does what it's supposed to do.
- **Reliability** – performs consistently without failure.
- **Usability** – easy to learn and use.
- **Efficiency** – optimal use of resources (time, memory).
- **Maintainability** – easy to modify/fix.
- **Portability** – works across different environments.
- Software quality is achieved through a combination of good design, coding standards, and rigorous testing (both verification and validation) throughout the development lifecycle.

14. Linguistics Metrics

Linguistic metrics are software metrics that measure a program's complexity or size based on **textual/syntactic properties** of the source code — such as the number of operators, operands, tokens, lines of code, etc. — without deeply analyzing control or data flow.

Key points:

- The most well-known linguistic metric is **Halstead's Software Science metrics**, based on counting:
  - n1 = number of distinct operators
  - n2 = number of distinct operands
  - N1 = total number of operators
  - N2 = total number of operands
- From these, Halstead derives measures like **Program Length, Vocabulary, Volume, Difficulty, and Effort**.
- **Weakness:** Linguistic metrics only consider the textual structure of code, not its logical/control complexity — so two programs of similar text size can have very different logical complexity (unlike structural metrics such as Cyclomatic Complexity).
- Used for estimating development effort, maintainability, and comparing code complexity.

---

15. Flow Graph / Flow Chart

A **flow graph** (control flow graph) is a directed graph representation of a program in which **nodes represent statements or blocks of sequential statements**, and **edges represent the flow of control** (branches, loops) between them.

Key points:

- Derived from a flow chart by combining sequential statements into single nodes and representing only decision points and junctions as separate nodes.
- A **flow chart** is a more detailed pictorial representation showing every step of a process/program using standard symbols (oval for start/end, rectangle for process, diamond for decision, arrows for flow).
- Flow graphs are the foundation for path testing, computing cyclomatic complexity, and identifying independent paths.
- Elements: **Nodes** (process/decision points), **Edges/Links** (control flow), **Entry node**, **Exit node**, **Regions**.
- Example: An `if-else` statement creates a decision node with two outgoing edges that merge at a junction node.

---

16. Data Flow Graph

A data flow graph is a graphical representation that shows how **data moves through a program**, focusing on where variables are **defined (created/assigned), used (referenced), and killed (destroyed/go out of scope)** — annotated onto the program's control flow graph.

Key points:

- Each node in the flow graph is annotated with data flow actions: **d** (define), **u** (use), **k** (kill) for each variable.
- Used in **data flow testing**, where test cases are designed to cover paths from a variable's definition to its use(s) — e.g., All-Defs, All-Uses, All-DU-Paths coverage criteria.
- Helps detect data flow anomalies such as: using a variable before defining it, defining a variable twice without using it in between, or killing a variable before using it.
- Especially useful for identifying bugs that pure control-flow testing might miss.

---

17. Types / Names of Bugs

Bugs can be broadly classified into the following types:

1. **Functional Bugs** – incorrect implementation of a specified function/requirement.
2. **Logic Bugs** – incorrect program logic causing wrong results (e.g., wrong loop condition).
3. **Syntax Bugs** – errors violating the language's grammar rules (usually caught by the compiler).
4. **Performance Bugs** – system fails to meet speed/efficiency requirements.
5. **Domain Bugs** – errors at input boundary conditions (off-by-one, boundary value errors).
6. **UI/Interface Bugs** – errors in how modules or the user interface communicate/display data.
7. **Processing Bugs** – errors that occur during computation/data processing steps.
8. **Test/Documentation Bugs** – incorrect or missing documentation/specifications leading to wrong implementation.

Understanding bug types helps testers design targeted test cases (e.g., boundary value analysis for domain bugs, interface testing for interface bugs).

---

18. Domain (Define)

In software testing, a **domain** is the set of all possible input values (or combinations of input values) that a program can accept, over which the program's behavior/output is defined.

Key points:

- **Domain testing** focuses on testing input values at and around the **boundaries** of these domains, since boundary values are the most error-prone areas.
- Each input variable has a valid range/domain; the program partitions the total input space into sub-domains, each mapped to specific processing/output behavior.
- Common domain errors: boundary shifted, boundary missing, boundary added incorrectly, closed/open boundary confusion (≤ vs <).

- Testing is done using techniques like **Boundary Value Analysis** and **Equivalence Partitioning**.
- Example: For an age field valid from 18–60, domain testing checks values like 17, 18, 60, 61.

---

19. Metrics (Define)

A software metric is a **quantitative measure** used to assess some property of software or the software development process, such as its size, complexity, quality, or the effort/productivity involved in producing it.

Key points:

- Metrics help in objectively evaluating and comparing software, predicting effort/cost, and identifying risky/complex modules that need more testing.
- **Types of metrics:**
  - **Structural/Complexity metrics** – e.g., Cyclomatic Complexity (based on control flow graph).
  - **Linguistic/Size metrics** – e.g., Halstead's metrics, Lines of Code (LOC).
  - **Quality metrics** – e.g., defect density, defect removal efficiency.
- **Desirable properties of a good metric:** should be simple to compute, objective, consistent, and should correlate well with the property being measured (e.g., complexity should correlate with error-proneness).

---

20. Transaction (Define)

A transaction is a **unit of work or a sequence of processing steps** initiated by a user or system that produces a specific result, treated as a single logical operation from start to completion. In transaction flow testing, it represents the path a piece of work takes through a system.

Key points:

- A transaction typically has a beginning, a series of processing steps, and an end (completion), and may involve multiple states/subsystems.
- Represented using a **transaction flow graph**, similar to a control flow graph, but showing the flow of a "unit of work" through the system rather than program statements.
- Testing transactions involves verifying each step executes correctly and the transaction completes with the expected outcome (or is correctly rolled back on failure).
- Example: An ATM cash withdrawal transaction — Insert card → Enter PIN → Select amount → Dispense cash → Print receipt.

---

21. Stable State

A stable state is a state in a state-transition system in which the system **remains indefinitely until an external input or event** occurs to trigger a transition to another state — i.e., it does not change on its own without an external stimulus.

Key points:

- Contrasts with **transient states**, which the system passes through momentarily before automatically moving to another state (without waiting for external input).
- Stable states represent "waiting" conditions in a system — e.g., an idle state waiting for user input.
- Important in state testing to correctly distinguish states that require an external trigger from those that are purely internal/automatic transitions.
- Example: In a traffic light, "Red" can be a stable state (waits for timer/pedestrian input) whereas a brief "transition" between colors may be a transient state.

Part – B

Software Testing - 5 Mark Answers

1. Domain Testing — Need, Concept, Model

Need: Most bugs occur at the boundaries of input domains rather than in the interior, so testing only "typical" values misses a large class of defects. Domain testing focuses effort where it statistically matters most, reducing test cases while increasing bug-detection efficiency.

Concept: A program's input space is divided into **domains** — sets of input values that cause the program to behave in a functionally equivalent way (i.e., execute the same path or produce the same class of output). Each domain is bounded by **borders**, defined by predicates (relational expressions) in the code, e.g., `x > 5`, `y <= 10`. Domain testing selects test points **ON** the border and **OFF** the border (just inside/outside) to detect boundary errors such as:

- Closure bugs (< used instead of <=)
- Shifted boundaries
- Missing/extra boundaries
- Tilted or parallel-boundary errors

Model: For an n-dimensional input space:

1. Identify all domains and the predicates defining their boundaries.
2. For a linear boundary, select **one ON point** and **one OFF point** for each boundary segment (open boundaries need the OFF point on the domain side that should be excluded; closed boundaries need it on the included side).

3. For each boundary defined by `n` variables, at least `n+1` ON points and 1 OFF point are needed to fully test a linear inequality boundary.
4. Combine points across all dimensions to build test cases; each domain requires at least one interior test point too.

Domain testing is most effective for numeric range checks, loop bounds, and decision predicates, and is usually combined with boundary value analysis and equivalence partitioning.

---

2. State Table / State Graph — Notes with Example

A **state graph** is a directed graph used to model software whose behavior depends on history (state machines, protocols, menu systems).

- **Nodes** = states of the system
- **Links (edges)** = transitions, each labeled with **input/event** that causes the transition and the **output/action** produced

A **state table** is the tabular equivalent — rows represent current states, columns represent input events, and each cell contains the next state (and output) for that state–input combination. It makes "impossible" or unspecified transitions immediately visible as blank cells.

Example — Simple Login System

States: `S0-Idle`, `S1-Waiting for Password`, `S2-LoggedIn`, `S3-Locked`

Current State \ Input	Enter Username	Correct Password	Wrong Password (3rd try)
S0-Idle	→ S1	–	–
S1-Waiting	–	→ S2	→ S3
S2-LoggedIn	–	–	–
S3-Locked	–	–	–

Uses:

- Detects unreachable states, dead states, and missing transitions.
- Basis for generating test cases: each transition (link) should be exercised at least once (state transition testing), and important transition **pairs/sequences** (n-switch coverage) should also be tested.
- Helps validate that every state has a defined response to every possible input (even if the response is "ignore").

---

3. Five Phases / Purpose of Testing (Beizer's Evolution of Testing)

Boris Beizer describes testing maturity as evolving through five phases, each with a different **purpose/mindset**:

1. **Phase 0 – Debugging Oriented:** No distinction between testing and debugging. Testing is informal; the purpose is just to make the program run without any conscious separate process.
2. **Phase 1 – Demonstration Oriented:** Purpose is to show that the program **works**, i.e., satisfies its requirements. Testing is done to demonstrate correctness, not to find faults — hence weak, biased test cases that avoid failure.
3. **Phase 2 – Destruction Oriented:** Purpose shifts to showing that the program **does not work** — testers actively try to break the software and expose failures. This is the beginning of a genuine testing mindset (find bugs, not prove absence of bugs).
4. **Phase 3 – Evaluation Oriented:** Purpose is to **evaluate/measure quality**, not simply to find bugs, using metrics like statement/branch coverage. Testing provides confidence measures and risk information alongside bug detection.
5. **Phase 4 – Prevention Oriented:** The most mature phase. Purpose is to **prevent bugs** from being introduced in the first place, through early test design (test-first, requirements-based test design), which reveals ambiguities and errors in specifications before code is written.

Overall purpose progression: run it → prove it works → try to break it → measure how good it is → stop bugs before they happen.

---

4. Five Rules of Path Product (Path Expression Algebra)

Path products are used in **flow-graph algebra** to symbolically represent all paths through a program graph so path expressions can be derived and simplified. The link weights (path names, e.g., a, b, c) are combined using multiplication (sequence) and addition (choice). Five algebraic rules govern simplification:

1. **Multiplication (Sequence) is not commutative but is associative:** $(ab)c = a(bc)$, but generally $ab \neq ba$ (order of traversal matters since paths are sequential).
2. **Distributive law over addition:** $a(b+c) = ab + ac$ and $(b+c)a = ba + ca$ — a path followed by a choice of two paths equals the choice of the two combined sequential paths.
3. **Addition (choice) is commutative and associative:** $a+b = b+a$ and $(a+b)+c = a+(b+c)$ — the order in which alternative paths are listed doesn't matter.
4. **Identity element rules:** Multiplying by the identity 1 leaves a path unchanged: $1a = a1 = a$. Adding the null path 0 (empty/non-existent path) leaves a path unchanged: $a+0 = a$.
5. **Loop rule (Kleene closure):** A loop that can execute zero or more times is represented as $a^* = 1 + a + a^2 + a^3 + \ldots = 1/(1-a)$ (in the algebraic sense),

i.e., the loop path product is the sum of all possible repetitions of the loop body, used to derive path expressions for graphs containing cycles.

These rules let a tester reduce a complex flow graph to a single simplified **path expression** representing every possible execution path.

---

5. Classify Metrics

Software metrics used in testing can be classified as follows:

A. Based on what is measured:

1. **Product Metrics** – measure attributes of the software product itself (size, complexity, quality). E.g., LOC, cyclomatic complexity, defect density, Halstead's metrics.
2. **Process Metrics** – measure the efficiency and effectiveness of the development/testing process. E.g., defect removal efficiency, test case execution rate, review effectiveness.
3. **Project Metrics** – measure project-level parameters like cost, schedule, resource utilization, productivity.

B. Based on measurement type:

1. **Size Metrics** – LOC, function points, number of modules.
2. **Complexity Metrics** – cyclomatic complexity (McCabe), Halstead complexity, nesting depth.
3. **Quality Metrics** – defect density, MTBF, reliability, defect severity distribution.
4. **Coverage Metrics** – statement coverage, branch/decision coverage, path coverage, condition coverage.
5. **Performance Metrics** – response time, throughput, resource utilization.

C. Based on direct/indirect measurement:

1. **Direct Metrics** – measured directly (e.g., LOC, execution time).
2. **Indirect (Derived) Metrics** – computed from direct metrics (e.g., defect density = defects/LOC, productivity = LOC/person-month).

Metrics guide test planning (where complexity is high, test more), estimate testing effort, and provide objective quality assessment rather than subjective judgment.

---

6. Importance of Bugs

Understanding and classifying bugs is central to testing because:

1. **Guides test design:** Knowing common bug categories (e.g., control-flow, data, boundary, interface bugs) helps testers design test cases that specifically target high-probability defect areas.
2. **Prioritization:** Not all bugs are equal — understanding bug severity/importance lets teams prioritize fixing critical (crash-causing, data-corrupting) bugs before minor cosmetic ones.
3. **Cost impact:** Bugs found early (requirements/design) are far cheaper to fix than those found in production — the importance of a bug is tied to *when* it's discovered, motivating early testing.
4. **Root cause analysis:** Classifying bugs by origin (requirement error, design error, coding error) helps identify weak points in the development process for future prevention.
5. **Risk assessment:** The importance of a bug depends on its **frequency of occurrence** and **consequence/severity** — a rare bug with catastrophic consequence (e.g., in aviation or medical software) may be more important than a frequent but trivial cosmetic bug.
6. **Quality metric basis:** Bug counts, density, and trends form key quality metrics used to decide release readiness.
7. **Improves reliability & trust:** Systematically valuing and tracking bugs increases software reliability, customer trust, and reduces maintenance costs.

In short, bugs are not just "things to fix" — their classification and importance analysis drive test strategy, process improvement, and risk management.

---

7. Structure / Structural Metrics

Structural metrics quantify the internal structure/complexity of software (as opposed to functional/black-box behavior), typically derived from the **control flow graph** or **program structure**.

Key Structural Metrics:

1. **Cyclomatic Complexity (McCabe's Metric):** `V(G) = E - N + 2P` where E = edges, N = nodes, P = number of connected components (usually 1). It equals the number of independent paths through a program and also equals (number of decision predicates + 1). Used to determine the minimum number of test cases needed for branch coverage.
2. **Halstead's Metrics:** Based on operator/operand counts:
    - Program length, vocabulary
    - Program volume (bits needed to represent the program)
    - Difficulty and effort metrics
    - Used to estimate error-proneness and maintenance effort.
3. **Nesting Depth / Structural Depth:** Measures how deeply loops/conditionals are nested — higher nesting increases cognitive complexity and testing difficulty.
4. **Fan-in / Fan-out:** Number of modules calling a given module (fan-in) and number of modules it calls (fan-out) — high fan-out indicates high complexity/coupling.

5. **Coupling and Cohesion metrics:** Measure inter-module dependency and intra-module relatedness.

Purpose: Structural metrics help identify complex, error-prone modules that require more rigorous testing, estimate testing effort, and set thresholds (e.g., "no module should exceed cyclomatic complexity 10") as part of quality standards.

---

8. Transaction Flow Testing Techniques

Transaction flow testing is used for systems that process **transactions** — a unit of work that flows through the system from entry to completion (e.g., an ATM withdrawal, an online order, a database update).

Steps/Techniques:

1. **Draw the Transaction Flow Graph:** Model the transaction's path through the system as a graph — nodes represent processing steps and links represent the flow of control/data between them.
2. **Identify Transaction Types:** Different transactions (e.g., deposit, withdrawal, balance inquiry) may follow different paths; each type is modeled and tested.
3. **Path Selection:** Select paths through the transaction flow graph covering:
    - All nodes (statement-level transaction coverage)
    - All links (each transition tested)
    - All critical/legal transaction paths, and select representative illegal ones.
4. **Sensitization:** Determine input data values needed to force the transaction down each selected path.
5. **Trace and validate output**: Execute the transaction and verify the output/state matches expected results at each processing stage.
6. **Multi-transaction / concurrent testing:** For systems with multiple simultaneous transactions, test interactions (e.g., race conditions, deadlocks, shared resource contention).

Benefits: Ensures each business-process path (not just each code path) is validated, catches integration-level errors between processing stages, and validates that the system behaves correctly under normal, boundary, and exceptional transaction scenarios.

---

9. Control Flow Graph (Flow Graph) — Elements

A **Control Flow Graph (CFG)** is a directed graph representation of all paths that might be traversed through a program during execution — the fundamental model used in structural/white-box testing.

Elements:

1. **Node (Circle):** Represents one or more sequential statements/process blocks with no branching (a "basic block"). Execution enters at the top and exits at the bottom with no branches in between.
2. **Edge/Link (Arrow):** Represents the flow of control from one node to another; corresponds to a possible transfer of control (branch, sequence, or jump).
3. **Decision Node (Predicate Node):** A node with two or more outgoing edges, representing a conditional statement (`if`, `while`, `case`) — the outcome of the predicate determines which edge is taken.
4. **Junction Node:** A node where two or more edges converge (e.g., after an if-else block or end of a loop).
5. **Region:** An area bounded by edges and nodes in a planar flow graph — the number of regions relates directly to cyclomatic complexity.
6. **Entry Node / Exit Node:** Single node representing program start; single (or normalized) node representing program end.
7. **Loop constructs:** Represented as a back-edge from a later node to an earlier node, forming a cycle in the graph.

Purpose: The CFG is the basis for path testing, statement/branch/condition coverage, cyclomatic complexity calculation, and identifying unreachable code or infinite loops.

---

10. Steps in Syntax Testing

Syntax testing is used to validate that a program correctly parses/handles input according to a formally defined grammar (BNF – Backus-Naur Form), commonly used for compilers, command interpreters, protocol parsers, and input validation routines.

Steps:

1. **Identify/Define the Input Grammar (BNF):** Formally specify the syntax rules of valid inputs using BNF or a similar grammar notation.
2. **Identify Syntax Elements:** Break down the grammar into terminals, non-terminals, and production rules covering all valid input formats.
3. **Generate Good (Valid) Test Cases:** Create test cases that conform to the grammar, covering every production rule at least once — ensures the parser accepts all legitimate input forms.
4. **Generate Bad (Invalid) Test Cases:** Deliberately create inputs that violate the grammar (missing tokens, wrong order, illegal characters, exceeding length limits) to verify the parser correctly **rejects** malformed input with appropriate error handling.
5. **Boundary Testing on Syntax Elements:** Test length limits, numeric ranges, and repetition limits at their boundaries (min, max, min-1, max+1).
6. **Combine and Mutate Fields:** Apply mutation-like techniques — swap, omit, duplicate, or corrupt tokens to generate a broad set of edge-case inputs.
7. **Automate Using a Syntax Test Generator/Tool:** Since the number of combinations is large, use tools that automatically generate test cases from the BNF grammar.
8. **Execute and Verify:** Confirm the system's parser produces correct output/AST for valid input and clean, informative errors (no crash) for invalid input.

Importance: Especially critical for security (input validation, injection prevention) and robustness of any input-parsing component.

---

11. Consequences / Types of Bugs

Types of Bugs (Beizer's Classification – by origin):

1. **Requirements & Specification Bugs** – incomplete, ambiguous, or incorrect requirements.
2. **Design Bugs** – incorrect algorithm, control flow, or data structure design.
3. **Coding Bugs** – syntax errors, logic errors, typos, incorrect use of operators.
4. **Data Bugs** – incorrect data definition, initialization, or type mismatches.
5. **Interface/Integration Bugs** – errors in communication between modules, incorrect parameter passing, protocol mismatches.
6. **Testing Bugs** – errors in the test cases or test environment itself, giving false pass/fail results.
7. **Documentation Bugs** – incorrect or missing documentation causing user/developer misunderstanding.

Consequences of Bugs (by severity/impact):

1. **Mild** – minor cosmetic issue, negligible impact (e.g., misaligned UI text).
2. **Moderate** – produces incorrect but recoverable results (e.g., wrong report formatting).
3. **Annoying** – e.g., legitimate names truncated, causing user irritation but not data loss.
4. **Disturbing** – valid transactions refused or system slows significantly.
5. **Serious** – loss or corruption of data, incorrect computations affecting business decisions.
6. **Very Serious** – wrong tasks executed instead of the requested one.
7. **Extreme** – arbitrary, seemingly random failures, hard to reproduce.
8. **Intolerable** – system unusable, long-term corruption not easily detected.
9. **Catastrophic** – system shuts down or causes total failure of dependent systems.
10. **Infectious** – bug corrupts other systems, spreads standards/data errors across a network of systems.

Significance: This classification helps prioritize fixes — catastrophic/infectious bugs demand immediate attention while mild bugs may be deferred, and it helps organizations understand risk exposure from each defect type.

---

12. Principles of State Testing

State (transition) testing verifies that a system with memory of past events behaves correctly. Core principles:

1. **Every state must be reachable:** Every defined state should be reachable from the initial state through some sequence of valid inputs; unreachable states indicate design flaws.
2. **Every state must have a defined exit:** No state should be a "trap" unless intentional (dead state); every state should allow transition to at least one other state (except designed terminal states).
3. **All transitions (links) should be exercised at least once:** Basic state coverage requires exercising every input/state combination defined in the state table, including verifying undefined transitions are properly rejected/handled.
4. **Test invalid/undefined transitions:** Verify the system correctly handles inputs not expected in a given state (should not crash or move to an undefined state).
5. **N-switch / transition-pair coverage:** Test sequences of 2, 3, or more consecutive transitions (not just single transitions) to catch bugs that only appear across a sequence of state changes.
6. **Guard conditions and outputs must be validated:** Not just the next-state, but also the output/action produced during each transition must be checked against expectation.
7. **Boundary and stress conditions on states:** Test state behavior under repeated/rapid transitions, and edge conditions like maximum stored history, timeouts, or simultaneous events.
8. **Return-to-initial-state validation:** Verify that resetting/initializing always returns the system to a known, correct state regardless of prior history.

These principles ensure both statically-defined behavior (state table correctness) and dynamically-emergent behavior (sequence-dependent bugs) are validated.

---

13. Steps in System Testing

System testing validates the **complete, integrated system** against specified requirements, in an environment resembling production. Typical steps:

1. **Requirement/Test Planning:** Review requirement specifications (functional & non-functional) and prepare the System Test Plan — scope, objectives, resources, schedule, entry/exit criteria.
2. **Test Environment Setup:** Configure hardware, software, network, and test data to mirror the production environment as closely as possible.
3. **Test Case Design:** Design test cases covering all functional requirements plus non-functional aspects: performance, security, usability, recovery, compatibility, load, and stress.
4. **Test Case Review:** Review designed test cases for completeness and correctness against requirements (traceability matrix).
5. **Test Execution:** Execute test cases systematically — functional testing first, then non-functional (performance, load, stress, security, usability, installation, recovery testing).
6. **Defect Logging & Tracking:** Log any deviations as defects in a bug tracking tool, with severity/priority classification.

7. **Defect Fixing & Retesting:** Developers fix reported defects; testers retest the fix and perform **regression testing** to ensure no new issues were introduced.
8. **Test Closure / Exit Criteria Verification:** Verify exit criteria are met (e.g., required pass rate, no open critical bugs), and prepare the **Test Summary Report**.
9. **Sign-off / Release Readiness:** Stakeholders review results and approve the system for the next stage (UAT / production release).

Types of tests performed within system testing: functional testing, performance testing, load/stress testing, security testing, usability testing, compatibility testing, recovery/failover testing, installation testing.

---

14. Interface Testing (Explain)

Interface testing verifies the correctness of communication and data exchange **between two or more components/modules/systems** — including software-to-software, software-to-hardware, and system-to-external-system interfaces (e.g., APIs, databases, third-party services).

Purpose: Even if individual modules work correctly in isolation, integration failures — wrong data formats, incorrect parameter order, timing mismatches, protocol errors — can cause system failure. Interface testing specifically targets these boundary/communication errors.

Types of Interfaces Tested:

1. **Module Interfaces** – parameter passing, return values, global data usage between internal functions/modules.
2. **API Interfaces** – request/response format, status codes, authentication, data types, error handling for internal or third-party APIs.
3. **Hardware-Software Interfaces** – device drivers, sensor data, I/O communication.
4. **User Interfaces** – how UI components exchange data with the backend (though often tested separately as UI testing).
5. **External System Interfaces** – file transfers, message queues, database connections between the system and external systems.

What is Checked:

- Correct data format, type, and range being passed across the interface.
- Correct sequencing/timing of calls (e.g., initialization before use).
- Proper error handling when the interface receives invalid/unexpected data (e.g., timeouts, malformed responses).
- Boundary conditions on data volume (large payloads, empty responses).
- Security of data exchange (encryption, authentication tokens).
- Behavior under interface failure (network down, service unavailable) — graceful degradation.

Approach: Typically done using **stub and driver** techniques during integration, or automated API testing tools (e.g., Postman) for live systems, verifying both the "happy path" and negative/error scenarios across the interface boundary.

Importance: Since most real-world defects arise at integration points rather than within a single module, interface testing is critical to overall system reliability.

Software Testing — 10 Mark Answers

1. Components of Decision Tables

A **decision table** is a structured tool used to represent complex combinations of conditions and their corresponding actions in a compact, tabular form. It is especially useful when a system's behavior depends on **multiple independent conditions** (business rules, complex logic), where plain text specifications become ambiguous.

Structure — Four Quadrants:

A decision table is divided into four quadrants, separated by a double line (horizontal and vertical):

```
 ---------------------------------
|   Condition Stub | Condition Entries |
|------------------|-------------------|
|   Action Stub    | Action Entries    |
 ---------------------------------
```

1. **Condition Stub (top-left):** Lists all the conditions/decision variables relevant to the problem (e.g., "Is customer a member?", "Order amount > $100?"). Each row is one condition.
2. **Condition Entries (top-right):** For each **rule** (column), shows the value (Y/N/True/False/Don't Care "–") that each condition takes. Each column represents one unique combination of conditions, called a **rule**.
3. **Action Stub (bottom-left):** Lists all possible actions the system can take (e.g., "Apply 10% discount", "Reject order", "Send confirmation email").
4. **Action Entries (bottom-right):** For each rule/column, marks (X or ✓) which actions are executed when that combination of conditions holds.

Additional Components:

- **Rules (Columns):** Each column is a complete rule — a unique combination of condition values mapped to the resulting actions. With `n` binary conditions, there can be up to `2^n` rules.
- **Don't Care Entries ("–"):** Indicate the condition's value is irrelevant to the outcome for that rule, allowing rule consolidation and reducing table size.
- **Impossible/Infeasible Rules:** Combinations of conditions that can never occur together are marked and excluded from testing.
- **Redundant/Contradictory Rules:** Identified during table verification — the same condition combination should never map to conflicting actions.

Example:

Conditions	R1	R2	R3	R4
Member?	Y	Y	N	N
Order > $100?	Y	N	Y	N
Actions				
10% Discount	X			
5% Discount		X		
No Discount			X	X

Use in Testing: Each rule (column) becomes at least one test case, ensuring every valid business-rule combination is exercised, and boundary/invalid combinations are also tested. Decision tables also help detect gaps (missing rules) and contradictions in the specification itself — thus serving both as a specification and a test-design technique.

---

2. Elaborate Model for Testing

The **testing model** conceptualizes software testing as an information-flow process comparing the actual behavior of a program against expected behavior, and is best represented by Beizer's **environment–program–bug** model, extended into a general test process model with the following elements:

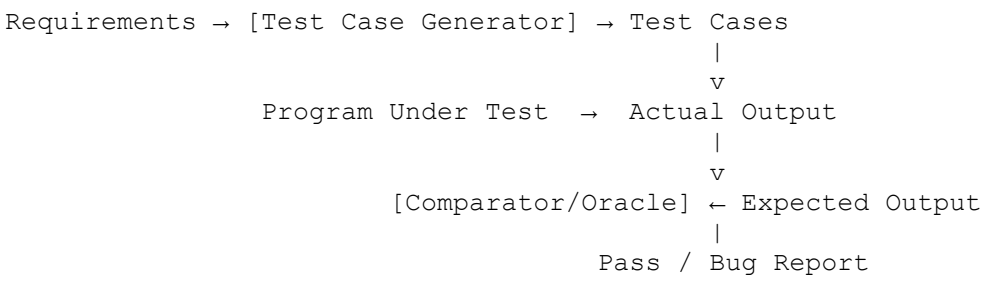
1. Sources of Information:

- **Requirements/Specifications** – the intended behavior.
- **The Program (Code)** – actual implementation.
- **Environment** – hardware, OS, network, external systems the program interacts with.

2. The Model Components (Six-part model):

1. **Testable Design / Requirement Model:** An abstraction of what the program *should* do, derived from requirements — forms the "expected" side of comparison.
2. **Program under Test:** The actual code, treated as a black box or white box depending on the technique used.
3. **Test Case Generator:** Uses the requirement model (and/or code structure) to generate a finite, practical set of test cases representing the (usually infinite) input domain.
4. **Test Execution Environment:** Executes the test cases against the program, capturing actual outputs.
5. **Comparator / Oracle:** Compares actual output to expected output (derived from the requirement model) and flags discrepancies as potential bugs.
6. **Feedback Loop:** Discrepancies are analyzed — if they represent real bugs, they're fixed and retested (regression); if the "expected" model itself was wrong, the model/requirements are corrected.

Flow Diagram (conceptual):



Key Principles Embedded in the Model:

- Testing can only show the **presence** of bugs, not their absence (Dijkstra's principle) — the model never claims complete correctness, only increases confidence.
- The **test oracle problem**: the model assumes we have a reliable way to determine expected output; when this is hard (e.g., complex numerical systems), oracle design itself becomes a challenge.
- **Exhaustive testing is impossible**, so the model emphasizes selecting a *representative, high-yield subset* of the input domain (linked to domain testing, boundary analysis, equivalence classes).

Elaboration with Beizer's Bug Model: Beizer models a bug as a deviation between the program's actual behavior and the expected behavior for some input, arising from an incorrect transformation of requirements into code. The testing model's job is to systematically search the input space to reveal such deviations with minimum test cases and maximum defect yield.

This general model underlies every specific testing technique (path testing, domain testing, state testing, etc.), each of which specializes how test cases are generated and how the oracle is constructed.

---

3. Transaction Flow Testing Techniques

Definition: Transaction flow testing is a **black-box, system-level testing technique** used for transaction-processing systems (banking, order processing, reservation systems) where a "transaction" is a unit of work that flows through several processing steps from initiation to completion.

Building the Transaction Flow Graph:

- **Nodes** represent processing steps (validate input, check balance, update database, generate receipt).
- **Links** represent the flow from one step to another, often labeled with the condition under which that path is taken.
- Distinguish **birth nodes** (transaction creation) and **death nodes** (transaction completion/termination).

Techniques / Steps:

1. **Transaction Flow Modeling:** Identify all transaction types the system supports and draw a flow graph for each, showing all decision points, parallel processing paths, and exception paths.
2. **Path Selection Strategies:**
    - **All-node coverage:** every processing step exercised at least once.
    - **All-link coverage:** every transition between steps exercised.
    - **All-path coverage** (where feasible): exercise all combinations, prioritizing critical business paths.
3. **Path Sensitization:** Determine specific input data that forces a transaction along each selected path (e.g., specific account balances to trigger "insufficient funds" path).
4. **Path Instrumentation / Tracing:** Insert tracking or logging to confirm the transaction actually followed the intended path during execution — validates test correctness itself.
5. **Testing Incomplete/Interrupted Transactions:** Simulate failures mid-transaction (crash, timeout, network failure) to verify rollback/recovery mechanisms (atomicity).
6. **Concurrent Transaction Testing:** Since multiple transactions may execute simultaneously, test for race conditions, deadlocks, and shared-resource contention.
7. **Volume/Load-based Transaction Testing:** Execute many transactions concurrently to validate throughput and stability under realistic loads.
8. **Negative Transaction Testing:** Submit invalid/malformed transactions to verify proper rejection and error messaging without corrupting system state.

Example: For an ATM withdrawal transaction: Insert Card → Validate PIN → Select Withdrawal → Check Balance → Dispense Cash → Print Receipt → Eject Card. Test paths include: valid withdrawal, insufficient balance, wrong PIN (3 attempts → card retained), network failure during dispense (must not lose money or leave inconsistent balance).

Importance: Since real business processes span multiple modules/subsystems, transaction flow testing catches integration and workflow-level defects that unit or pure code-path testing would miss, and it directly validates business-critical use cases end-to-end.

---

4. Data Flow Testing Strategies

Concept: Data flow testing is a white-box technique that focuses on the **life cycle of data (variables)** — from where they are **defined (d)**, to where they are **used (u)**, to where they are **killed/undefined (k)** — rather than purely on control flow. It's based on the idea that many bugs occur in the interaction between data definition and use (e.g., using a variable before initializing it, or never using a computed value).

Key Definitions:

- **Define (d):** A statement where a variable is assigned a value.
- **Use (u):** A statement where a variable's value is read/referenced. Two sub-types:
    - **c-use (computational use):** used in computation/expression.
    - **p-use (predicate use):** used in a decision/condition.
- **Kill:** A variable's value becomes undefined (goes out of scope, or is reassigned without being used).

- **DU path (Definition-Use path):** A path from a variable's definition to a use, with no redefinition of that variable in between.
- **DU chain / DU pair:** The pairing of a specific definition with a specific subsequent use.

Data Flow Anomalies (bugs targeted):

1. Variable defined but never used (dead code / wasted computation).
2. Variable used before being defined (uninitialized variable).
3. Variable defined twice without use in between (redundant/possibly erroneous assignment).

Testing Strategies (Coverage Criteria, from weakest to strongest):

1. **All-Defs Coverage:** For every definition of a variable, at least one path to *some* use of it is tested.
2. **All-Uses Coverage:** For every definition, test paths to **every** use (both c-use and p-use) reachable from that definition.
3. **All-du-paths Coverage:** For every definition-use pair, test **all** distinct (loop-free) paths connecting them — the strongest and most thorough criterion.
4. **All-c-uses/Some-p-uses & All-p-uses/Some-c-uses:** Intermediate strategies prioritizing either computational or predicate uses, with minimal coverage of the other type.
5. **All-Predicate-Uses Coverage:** Ensures every use of a variable in a decision/branch condition is tested — important since predicate variables control program flow.

Relative Strength: All-du-paths ⊇ All-uses ⊇ All-c-uses/Some-p-uses ≈ All-p-uses/Some-c-uses ⊇ All-defs ⊇ All-edges (branch coverage) ⊇ All-nodes (statement coverage).

Example:

```
1: x = 10          // def(x)
2: if (x > 5)      // p-use(x)
3:    y = x + 1    // c-use(x), def(y)
4: print(y)        // c-use(y)
```

All-defs requires testing at least one path from `x`'s definition (line 1) to a use (line 2 or 3); All-uses requires testing paths to both line 2 (p-use) and line 3 (c-use).

Importance: Particularly effective for catching bugs in variable initialization, loops, and complex data manipulation logic that pure control-flow (branch) testing can miss, since data flow testing is sensitive to *how data moves*, not just *which branches execute*.

---

5. Linguistic Metrics (Discuss)

Linguistic metrics measure software complexity based on treating source code as a "language" composed of **operators** and **operands**, most notably formalized in **Halstead's Software Science**.

Basic Counts:

- `n1` = number of distinct (unique) operators
- `n2` = number of distinct (unique) operands
- `N1` = total occurrences of operators
- `N2` = total occurrences of operands

Derived Metrics:

1. **Program Vocabulary:** `n = n1 + n2`
2. **Program Length:** `N = N1 + N2`
3. **Estimated Length:** `N^ = n1 log₂n1 + n2 log₂n2` (theoretical length based on vocabulary)
4. **Program Volume:** `V = N × log₂n` — represents the number of mental comparisons/bits needed to generate the program; a measure of program size in information-theoretic terms.
5. **Program Difficulty:** `D = (n1/2) × (N2/n2)` — reflects how hard the program is to write/understand (more distinct operators and repeated operand use increase difficulty).
6. **Programming Effort:** `E = D × V` — estimated mental effort (in elementary discriminations) needed to implement the program.
7. **Time to Program:** `T = E / β` (β ≈ 18, Stroud number) — estimated time in seconds.
8. **Estimated Number of Bugs (Delivered Bugs):** `B = V / 3000` — an empirical estimate of the number of bugs likely present, based on the idea that programmers make an error roughly every 3000 mental discriminations.

Discussion:

- Linguistic metrics are purely **syntactic** — they don't consider logic complexity (unlike cyclomatic complexity, which is structural), so they are best used **alongside** structural metrics, not as a replacement.
- **Strengths:** Language-independent counting rules, easy to automate via static analysis/parsing, provides quick early estimates of size, effort, and defect-proneness before testing begins.
- **Weaknesses:** Doesn't account for control flow complexity, can be gamed by verbose/terse coding styles, empirical constants (like the 3000 divisor) are dataset-specific and may not generalize across languages/domains.
- **Use in Testing:** Modules with high Halstead volume/effort scores are flagged as higher risk and prioritized for more rigorous testing; the estimated bug count helps set realistic defect-detection targets during test planning.

Overall, linguistic metrics provide an information-theoretic lens on code complexity, complementing structural (graph-based) metrics to give testers a fuller picture of where defects are likely to hide.

---

6. Testing vs Debugging (Compare)

Aspect	Testing	Debugging
Purpose	To find/detect the presence of defects (failures) in software	To locate the root cause of a known defect and fix it
Starting Point	Begins with a known/expected requirement, executed against unknown behavior	Begins with a known failure/symptom, and searches for the unknown cause
Performed By	Typically done by testers/QA engineers (independent of developer, to avoid bias)	Typically done by developers who wrote/understand the code
Nature of Activity	A planned, systematic process — can be scripted, automated, and repeated	An unplanned, investigative, often ad-hoc / creative activity
Input	Test cases derived from requirements/specifications	A bug report / failure observed during testing or in production
Output	A list of defects/failures found, pass/fail status, coverage reports	A code fix / patch that removes the root cause
Timing	Occurs throughout the SDLC, especially before release, and continues post-release (regression)	Occurs after a failure has been identified, usually during/after coding and after testing reports a bug
Can be Automated?	Highly automatable (regression suites, CI pipelines)	Difficult to fully automate; requires human reasoning, though tools (debuggers, profilers) assist
Knowledge Required	Requires understanding of requirements/specifications and user perspective	Requires deep understanding of code, algorithms, and system internals
Approach	Black-box or white-box; focuses on *what* the system does	Always white-box; focuses on *why* the system does it
Tools Used	Test management tools, automation frameworks (Selenium, JUnit), bug trackers	Debuggers (gdb, IDE breakpoints), profilers, log analyzers, static analysis tools
Deliverable	Test summary report, defect log	Fixed source code, root-cause analysis document
Relation	Testing *reveals* that a bug exists	Debugging *removes* the bug that testing revealed
Mindset	Destructive — trying to break the system	Constructive — trying to repair the system
Cost impact if skipped	More defects reach production, higher risk	Bug remains unresolved, defect persists in the codebase

Summary: Testing and debugging are complementary but distinct activities in the quality assurance lifecycle. Testing is a *verification* activity that answers "does a defect exist?", while debugging is a *diagnosis and correction* activity that answers "where exactly is it, and how do we fix it?" A mature process keeps them separate — testers should not debug, and developers debugging their own code should not also be its sole tester — to maintain objectivity and thoroughness.

7. Concept of Path Testing

Definition: Path testing (also called **basis path testing**) is a white-box testing technique where test cases are designed to execute **specific independent paths** through a program's control flow graph, ensuring every linearly independent path is exercised at least once.

Foundational Concepts:

1. **Control Flow Graph (CFG):** The program is first represented as a directed graph — nodes are sequential statement blocks, edges represent control transfer, and decision nodes have multiple outgoing edges.
2. **Path:** A sequence of nodes/edges from the entry node to the exit node of the graph.
3. **Independent Path:** A path that introduces at least one new edge (a new statement or condition) not covered by previously selected paths — this ensures no redundant test cases.
4. **Cyclomatic Complexity `V(G)`:** Determines the *minimum number* of independent paths required, calculated as: `V(G) = E - N + 2` (E = edges, N = nodes), or equivalently `V(G) = P + 1` (P = number of decision predicates), or `V(G) = number of regions` in the planar graph.

Steps in Path Testing:

1. Draw the program's control flow graph from source code.
2. Compute cyclomatic complexity `V(G)` to determine the number of test cases needed.
3. Identify a **basis set** of `V(G)` linearly independent paths.
4. Derive input data (test cases) that force execution along each of these paths.
5. Execute the test cases and verify actual results against expected results.
6. Ensure every path in the basis set has been executed at least once (this guarantees every edge and node is covered, since the basis set is a spanning set for the graph's cycle space).

Example:

```
1: read(x)
2: if (x > 0)
3:    y = x * 2
4: else
5:    y = -x
6: print(y)
```

CFG has one decision node (line 2) → `V(G) = 2` (2 independent paths).

- **Path 1:** 1→2→3→6 (x > 0 branch)
- **Path 2:** 1→2→5→6 (x <= 0 branch)

Test cases: `x = 5` (expect y=10), `x = -3` (expect y=3).

Advantages:

- Provides a measurable, minimum-yet-sufficient set of test cases (avoids both under-testing and redundant over-testing).
- Guarantees every statement and branch is executed at least once.
- Directly tied to a quantifiable complexity metric, useful for both test planning and code-quality assessment (modules with high `V(G)` are harder to test and more error-prone).

Limitations: Doesn't guarantee all *combinations* of conditions are tested (only linear independence, not full path coverage for graphs with loops or nested conditions), and cannot detect data-related bugs not tied to control flow (which is where data flow testing complements it).

---

8. Steps of Syntax Testing

(See detailed explanation)

Definition: Syntax testing validates that a system correctly accepts inputs conforming to a formally defined grammar and correctly rejects inputs that violate it — critical for compilers, parsers, command-line interfaces, protocol handlers, and any input-validation logic.

Detailed Steps:

1. **Define/Obtain the Formal Grammar (BNF):** Express the valid input syntax using **Backus-Naur Form** — a set of production rules composed of terminals (literal tokens) and non-terminals (composed of other rules). Example: `<command> ::= <verb> <object>` `<verb> ::= "GET" | "SET"`.
2. **Identify All Syntax Elements:** Break the grammar down into all terminal symbols, non-terminal symbols, and the production rules connecting them — build a complete inventory of the "vocabulary" and "grammar rules" to be tested.
3. **Design "Good" (Valid) Test Cases:** For each production rule, create at least one test case that satisfies it, ensuring every legal syntactic construct in the grammar is exercised at least once (production rule coverage).
4. **Design "Bad" (Invalid) Test Cases:** Systematically violate the grammar — omit required tokens, add illegal tokens, use wrong token order, use out-of-range values, exceed length limits — to verify the system correctly detects and rejects invalid input, without crashing.
5. **Test Boundary Conditions of Syntax Elements:** For numeric or length-bound fields defined in the grammar (e.g., max field length, numeric ranges), apply boundary value analysis: min, min−1, max, max+1.
6. **Apply Mutation/Perturbation to Valid Inputs:** Take valid inputs and mutate them (swap tokens, duplicate tokens, truncate, insert special/illegal characters) to generate a large, systematic set of edge cases efficiently.
7. **Automate Test Case Generation:** Because the combinatorics of grammar rules can be huge, use (or build) a **syntax test case generator** tool that parses the BNF and auto-generates good/bad test cases.
8. **Execute Test Cases Against the Parser/Input Handler:** Run all generated test cases and record actual behavior (accepted / rejected / crashed).

9. **Verify Error Handling Quality:** For every rejected (bad) input, confirm the system produces a clear, correct error message/code rather than an ambiguous failure, security leak, or crash.
10. **Regression on Grammar Changes:** Whenever the grammar (input format/protocol) changes/version-updates, re-run and extend the syntax test suite to cover new/modified rules.

Importance: Syntax testing is particularly vital for security (preventing injection attacks, buffer overflows from malformed input) and robustness, since parsers are common attack surfaces and a frequent source of critical field-proven bugs.

---

9. Logic-Based Testing with Examples

Definition: Logic-based testing is a white/black-box technique that derives test cases directly from the **Boolean logic conditions** governing decisions in a program or specification, ensuring that all meaningful combinations of logical conditions are exercised — not just the two outcomes (true/false) of a compound decision.

Key Techniques:

A. Decision (Predicate) Coverage: Each decision evaluates to True at least once and False at least once. (Weak — doesn't test individual conditions within a compound predicate.)

B. Condition Coverage: Each individual condition within a compound predicate takes both True and False at least once, regardless of the overall decision outcome.

C. Decision/Condition Coverage: Combines both — every decision and every individual condition takes both outcomes.

D. Modified Condition/Decision Coverage (MC/DC): The gold standard (used in safety-critical/avionics software, DO-178B). Requires:

- Every condition takes both True/False.
- Every decision takes both True/False.
- Each condition is shown to **independently** affect the decision's outcome (i.e., toggling that one condition alone, while others stay fixed, changes the overall decision result).

E. Multiple Condition (Combinatorial) Coverage: All possible combinations `2^n` of `n` conditions in a compound predicate are tested.

F. Cause-Effect Graphing: Causes (input conditions) and effects (output actions) are modeled as a Boolean graph with AND/OR/NOT relationships, then converted into a decision table to systematically derive test cases covering all logical combinations.

Example:

Consider the compound condition: `if (A AND B) OR C then action`

Conditions: A, B, C (3 boolean conditions)

Test	A	B	C	(A∧B)∨C	Coverage Purpose
T1	T	T	F	T	Decision True via A∧B
T2	F	T	F	F	Decision False, isolates A's effect
T3	T	F	F	F	Isolates B's effect
T4	F	F	T	T	Decision True via C alone, isolates C's effect

This test set achieves **MC/DC**: each of A, B, C is independently shown to affect the outcome (T1 vs T2 isolates A; T1 vs T3 isolates B; comparing a False case to T4 isolates C).

Cause-Effect Graph Example: For a form requiring "Age between 18-60 AND (Has ID OR Has Passport)":

- Causes: C1 = Age valid, C2 = Has ID, C3 = Has Passport
- Effect: E1 = Application Accepted
- Logic: `E1 = C1 AND (C2 OR C3)` This is converted to a decision table listing all combinations, and each column becomes a test case.

Importance: Logic-based testing (especially MC/DC) is essential in safety-critical domains (aerospace, medical devices, automotive) because simple decision/branch coverage can miss serious logic errors hidden inside compound conditions — e.g., a bug in condition B might never surface if it's always masked by condition A being True.

---

10. Domain Testing Model

Concept Recap: Domain testing treats the input space of a program as being partitioned into **domains** by the **predicates** in the code (comparisons like `x < 10`, `y >= 0`). Each domain corresponds to a distinct path/processing behavior. Bugs cluster at **domain boundaries**, so the model focuses test-point selection there.

The Domain Testing Model — Formal Structure:

1. **Domain Boundary:** Defined by a linear inequality of the form `a1x1 + a2x2 + ... + anxn ⊕ c`, where ⊕ is a relational operator (<, ≤, >, ≥, =, ≠).
2. **Closed vs Open Boundary:**
   - **Closed boundary** (≤, ≥, =): the boundary line itself belongs to the domain being tested.
   - **Open boundary** (<, >): the boundary line does *not* belong to the domain.
3. **ON Points and OFF Points:**
   - **ON point:** A point lying exactly on the boundary.
   - **OFF point:** A point slightly off the boundary — placed on the side that should be **excluded** if the boundary is closed, or the side that should be **included** if the boundary is open. (I.e., OFF point is always chosen on the "other"

domain's side relative to what's ON the tested domain, to maximize the chance of detecting an off-by-one/closure bug.)

4. **Number of Test Points:** For an `n`-dimensional boundary (defined by `n` variables), domain testing theory shows that `n+1` ON points and `1` OFF point are sufficient to test that boundary segment for the seven classic boundary bug types (shifted, tilted, missing, extra boundary, etc.).
5. **Bug Types Detected by the Model:**
    - **Closure bugs:** operator should be `<=` but is coded as `<` (or vice versa).
    - **Shifted boundary:** the boundary constant is off (e.g., `x < 10` should be `x < 12`).
    - **Tilted boundary:** the boundary's slope/coefficients are wrong.
    - **Missing boundary:** a needed boundary condition doesn't exist in code (domains incorrectly merged).
    - **Extra boundary:** an unnecessary boundary splits a domain that should be single.
6. **Procedure to Apply the Model:** a. Identify all domains and their boundary predicates from the code/specification. b. For each boundary, select ON points (n+1 of them, spread along the boundary) and 1 OFF point. c. Combine ON/OFF points across all boundaries relevant to a given domain to form complete test cases (all other variables held at "typical"/interior values). d. Execute and check that each point lands in the correct domain and produces correct output. e. Add interior points for each domain to also test typical-case behavior, not just boundaries.

Example: For domain defined by `x >= 0 AND x <= 100`:

- Boundary 1 (`x=0`, closed, lower): ON = 0, OFF = -1
- Boundary 2 (`x=100`, closed, upper): ON = 100, OFF = 101

Significance: This model formalizes what would otherwise be ad-hoc "boundary value analysis" into a rigorous, provably sufficient test-point selection strategy grounded in linear algebra/geometry of the input space, making it especially powerful for numeric, range-based, and multi-variable decision logic.

---

11. State Graphs (Detailed)

Definition: A state graph (finite state machine model) represents a system whose output/behavior depends not just on current input but also on the **history** of previous inputs (i.e., systems with "memory"). It's widely used for protocol design, embedded systems, GUI navigation, menu-driven systems, and workflow engines.

Components (detailed):

1. **States (Nodes):** Represent distinct configurations/modes of the system. Types:
    - **Initial state:** system's starting condition.
    - **Final/Terminal state:** end of processing (may or may not exist, depending on system).
    - **Intermediate states:** all others.

2. **Transitions (Directed Edges):** Represent a change from one state to another triggered by an **input/event**, and may produce an **output/action**. Notation is often `Input/Output` on the edge label.
3. **Self-loop:** A transition that starts and ends at the same state (input received but state doesn't change, though an action may still occur).
4. **Guard Conditions:** Additional Boolean conditions attached to a transition that must be true for the transition to fire (beyond just receiving the input event).

State Table (tabular equivalent): Rows = current states, Columns = input events, Cells = (next state, output). Useful to spot missing transitions (blank cells) systematically — harder to see in graph form for large systems.

Properties Verified Using State Graphs:

1. **Reachability:** every state must be reachable from the initial state.
2. **Completeness:** every state should have a defined response (even if "ignore" or "error") for every possible input — undefined transitions are a common source of bugs.
3. **No unintended dead states:** unless designed as terminal, no state should trap the system.
4. **Determinism:** for a deterministic FSM, a given (state, input) pair should map to exactly one next state.

Test Coverage Criteria based on State Graphs:

1. **State Coverage (Node coverage):** every state visited at least once.
2. **Transition Coverage (Link/Branch coverage — 0-switch):** every transition/edge exercised at least once — this is the most common minimum criterion.
3. **N-switch coverage:** every sequence of N consecutive transitions is tested (e.g., 1-switch tests every pair of consecutive transitions), catching bugs that emerge only across a sequence, not a single transition.
4. **Path/Sequence Coverage:** test specific important sequences of transitions corresponding to real use-case scenarios (e.g., login → browse → checkout → logout).
5. **Guard Condition Coverage:** each guard condition tested both true and false where relevant.

Example — Traffic Light Controller:

States: `Red`, `Green`, `Yellow`

Current \ Input	**Timer Expires**
Red	→ Green
Green	→ Yellow
Yellow	→ Red

Test cases: verify each transition (Red→Green, Green→Yellow, Yellow→Red) occurs correctly on timer expiry, verify no other input causes a transition, and verify the full cycle (Red→Green→Yellow→Red) repeats correctly (2-switch/sequence coverage).

Why Detailed State Testing Matters: Bugs in stateful systems are often **history-dependent** — e.g., a bug that only appears if a user logs out and logs back in a specific number of times, or a race condition only visible across a sequence of 3+ transitions. Simple transition coverage (0-switch) misses these; N-switch and scenario-based coverage are needed for thorough validation of realistic, mission-critical stateful systems (e.g., payment gateways, aircraft control software, network protocols).

---

12. Domains and Testability / Interface

Domains and Testability:

Testability refers to how easily a program's correctness can be verified via testing — i.e., how observable and controllable its behavior is. The **domain structure** of a program directly affects its testability:

1. **Well-defined, non-overlapping domains** (clear, simple boundary predicates) → high testability, since ON/OFF boundary points can be easily derived and each domain's expected behavior is unambiguous.
2. **Overlapping or ambiguous domains** (where two different predicates could both be satisfied, leading to undefined/ambiguous behavior) → poor testability, since it's unclear which processing path *should* execute, making the oracle (expected result) hard to define.
3. **Domain closure bugs and testability:** If a domain's boundary logic uses the wrong relational operator (< vs <=), the domain becomes harder to test predictably, since the "correct" location of the boundary itself is ambiguous without going back to requirements.
4. **High-dimensional domains** (many variables defining a boundary) reduce testability because the number of ON/OFF points needed grows (n+1 ON points per boundary for n variables), and combining multiple boundaries multiplies test case counts — testability decreases as domain complexity increases.
5. **Domains and equivalence partitioning:** Good domain design (aligned with equivalence classes) improves testability by allowing a small number of representative points per domain to stand in for the whole domain's behavior — poor domain design (where behavior is inconsistent even within a "domain") breaks this assumption and lowers testability.

Interface and Testability:

The **interface** between modules/components is a major factor in overall system testability:

1. **Observability:** A testable interface must allow testers to observe outputs/internal states easily (e.g., via logs, return values, monitoring hooks) — poor interfaces that hide state (e.g., only side effects with no return values) reduce testability.
2. **Controllability:** A testable interface must allow testers to easily set up any required input/state combination — interfaces requiring complex, hard-to-reproduce setup (e.g., deeply nested hidden state) reduce testability.

3. **Interface Complexity and Domain Testing:** Interfaces often define the "boundary" between domains of two different modules — if module A's output domain doesn't exactly match module B's expected input domain (a **domain mismatch**), integration bugs occur (e.g., Module A allows `x = 0` but Module B's logic assumes `x >= 1`).
4. **Interface Contracts:** Clear, formally specified interface contracts (types, valid ranges, pre/post-conditions) act like domain boundary definitions at the module level — well-specified contracts make interface testing tractable (test the boundary of the contract, just like a domain boundary); poorly specified/implicit contracts make interface bugs likely and hard to detect via testing.
5. **Improving Testability via Interface Design:** Adding explicit validation at interface boundaries, using strongly typed parameters, avoiding hidden global state, and providing test hooks/stubs all directly increase both domain clarity and interface testability.

Conclusion: Testability isn't just a property of code correctness — it's shaped by how clearly domains are defined and how observable/controllable the interfaces between components are. Testing strategy (domain testing at boundaries, interface/stub-driver testing at module seams) is most effective, and cheapest, when systems are designed from the start with clear domain boundaries and well-specified, observable interfaces.

---

13. Three Distinct Kinds of Testing

Software testing can be broadly classified into **three fundamentally distinct kinds**, based on *what aspect of the system* is being validated:

1. Functional Testing (Requirements-Based / Black-Box Testing)

- **Focus:** Validates *what* the system does — whether it correctly implements the specified functional requirements, regardless of internal implementation.
- **Approach:** Black-box — test cases derived purely from specifications/requirements, without looking at source code.
- **Techniques:** Equivalence partitioning, boundary value analysis, decision tables, state transition testing, use-case based testing.
- **Goal:** Confirm the system behaves as the user/requirements expect for both valid and invalid inputs.
- **Example:** Verifying a login form accepts valid credentials and correctly rejects invalid ones.

2. Structural Testing (Code-Based / White-Box Testing)

- **Focus:** Validates *how* the system does it — the internal logic, control flow, and data flow of the code itself.
- **Approach:** White-box — test cases derived from the source code's structure (control flow graph, data flow graph).
- **Techniques:** Statement coverage, branch/decision coverage, path testing (cyclomatic complexity), condition coverage (MC/DC), data flow testing (def-use paths).

- **Goal:** Ensure all code paths, branches, and logic combinations are exercised, catching implementation errors (dead code, logic mistakes, uninitialized variables) that functional testing (which only checks external behavior) might miss.
- **Example:** Ensuring both branches of an `if-else`, and all independent paths through a function, are executed at least once.

3. Non-Functional Testing (Quality Attribute Testing)

- **Focus:** Validates *how well* the system performs — quality attributes beyond pure functional correctness.
- **Approach:** Often black-box, but requires specialized environments/tools (load generators, security scanners).
- **Sub-types:**
  - **Performance Testing:** response time, throughput under load.
  - **Security Testing:** vulnerability to unauthorized access, injection attacks.
  - **Usability Testing:** ease of use, user experience.
  - **Reliability/Recovery Testing:** behavior under failure and recovery correctness.
  - **Compatibility Testing:** behavior across different OS/browsers/devices.
- **Goal:** Ensure the system meets non-functional requirements (SLAs, regulatory, UX standards) that functional correctness alone doesn't guarantee.
- **Example:** Verifying the system handles 10,000 concurrent users within a 2-second response time, or that user passwords are stored encrypted.

Why These Three Are "Distinct": Each answers a fundamentally different question about the same system:

- Functional Testing → "Does it do the right thing?"
- Structural Testing → "Is the internal logic correctly and completely exercised?"
- Non-Functional Testing → "Does it do the right thing well enough, under real-world conditions?"

A program can pass all functional tests and even achieve 100% structural (code) coverage, yet still fail under load, be insecure, or be unusable — which is why comprehensive quality assurance requires **all three kinds** of testing to be applied together, at different stages of the SDLC (structural testing typically at unit level, functional at integration/system level, non-functional mostly at system/acceptance level).