1. Test Design Techniques (Black-box & White-box)

These are ways to design test cases systematically.

- **Equivalence Partitioning (EP):**
  Divide inputs into groups ("partitions") where test results are expected to behave the same.
  Example: Age input (1–100). Test just one value from each range: valid (25), invalid low (0), invalid high (101).
- **Boundary Value Analysis (BVA):**
  Defects often occur at edges of ranges. Test min, max, and just inside/outside boundaries.
  Example: If valid age is 18–60 → test 17, 18, 60, 61.
- **Decision Tables:**
  Useful when system behavior depends on multiple conditions.
  Example: ATM → card valid/invalid × PIN correct/incorrect. Make a table with all combinations.
- **State Transition Testing:**
  Based on system states and events that trigger changes.
  Example: Login system: Locked → Unlocked → Locked (depending on correct/incorrect attempts).
- **White-box basics (Statement & Branch Coverage):**
  - **Statement Coverage:** Ensure every line of code executes at least once.
  - **Branch Coverage:** Ensure each decision (if/else) executes both true & false paths.

---

2. Test Management & Estimation

- **Three-Point Estimation (PERT formula):**
  Uses: Optimistic (O), Most likely (M), Pessimistic (P).
  Formula = (O + 4M + P) ÷ 6.

- **Wideband Delphi:**
  Think of this as a **group-based estimation method**.
  Steps (simple version):
  1. A **moderator** shares the task (e.g., "How long will login testing take?").
  2. Each **expert gives their estimate** individually (e.g., 2 days, 4 days, 3 days).
  3. Everyone's estimates are **shown anonymously** (so no one feels pressured).
  4. The team **discusses why their numbers are high/low**.
  5. They **re-estimate again**.
  6. This repeats until the estimates become **close to each other**.

  *Purpose: To get a **realistic estimate** that uses multiple perspectives, instead of one person guessing.*

- **Prioritization (Risk-based):**
  - We can't always test everything, so we decide **what to test first**.
  - The rule: **Test the most important or risky features first.**

  How do we know "risky"?

  - **High business impact** → If it fails, it hurts the customer a lot.
  - **High technical risk** → Complex code, new technology, or many integrations.
  - **Dependencies** → Something must work before others can be tested.

  Example:
  In an e-commerce app:
  - Payment → **High risk** (if it fails, business loses money). Test this first.
  - Search bar → **Medium risk**.
  - Product reviews → **Low risk**, can be tested later.

  Purpose: Save time and reduce chances of missing critical bugs.

- **Test Progress Monitoring (metrics):**
  This is about **tracking how testing is going** using measurable numbers.
  The common metrics:
    - Shows current quality.
    - Example: Out of 100 test cases, 70 passed, 20 failed, 10 not run → 70% passed.

  2. **Defect Density**
     - How many bugs per "size" of software.
     - Formula: Defects ÷ Size (e.g., per 1,000 lines of code, or per feature).
     - Example: If module A has 10 bugs in 1,000 lines and module B has 20 bugs in 500 lines → module B is riskier.
  3. **% Completed vs Planned**
     - Are we on track with the test plan?
     - Example: Planned to finish 50 test cases today, but only 30 done → 60% completion.

  Purpose: These numbers help managers see progress, detect delays, and decide if the product is ready.

In short:
- **Wideband Delphi** → A group method for better estimates.
- **Risk-based Prioritization** → Test high-risk/impact features first.
- **Monitoring with Metrics** → Use numbers to check testing health and progress.

---

3. Defects, Retesting, and Regression

- **Defect Life Cycle:** New → Assigned → Fixed → Retested → Closed.
- **Confirmation Testing (Retesting):** Check if a specific defect was fixed.
- **Regression Testing:** Check if new changes broke existing functionality. Essential in Agile/CI.

---

4. Test Levels & SDLC Alignment

- **Unit Testing:** Done by developers (testing smallest code units).
- **Integration Testing:** Verify modules work together.
- **System Testing:** Test complete application (functional + non-functional).
- **Acceptance Testing:** Done by end-users/customers to validate business needs.

Who tests what:

- Developers → Unit & sometimes Integration.
- Testers → Integration, System.
- Users/clients → Acceptance.

Summary Table

Level	Scope	Who tests it?	Example (Software)	Example (Car)
Component Testing	Smallest piece (unit)	Developer	Password validation function	Engine alone
Component Integration	Components together	Dev/Testers	Login form + database	Engine + gearbox
System Testing	Whole system	Testers	Entire e-commerce app	Whole car drive
System Integration	System + external systems	Testers	Payment gateway integration	GPS with Google Maps
Acceptance Testing	Business/user needs	Users/Clients	Final check by client	Customer test-drive

5. Agile Testing (if part of syllabus version)

- **Tester's Role in Agile:** Part of the development team, collaborate closely with developers and business.
- **Test Automation:** Used heavily in Agile for quick feedback.
- **Continuous Integration (CI):** Code is integrated/tested frequently.
- **TDD (Test Driven Development):** Write tests *before* code.
  The cycle:
  - Example: "Login function should return TRUE when username and password are correct."
  - Run the test → it **fails** (because no code yet).
  - Write the minimum code to **make the test pass**.
  - Refactor (clean code) while keeping the test green.

  Purpose: Ensure the code is designed to pass clear tests from the start → fewer bugs later.

- **ATDD (Acceptance Test Driven Development):** Acceptance criteria written as tests before implementation.

  Example: For an online shopping cart:

  - Acceptance criterion: "System should calculate total price including tax."
  - Acceptance test (in plain language):
    - Given 2 items costing $10 each and tax is 10%
    - When I checkout
    - Then the total should be $22

  ATDD makes sure developers build exactly what the **business expects**.

Key Takeaways

- **Tester's role** → part of Agile team, collaborates early, ensures quality at every step.
- **Automation** → critical for speed; regression & CI rely on it.
- **Continuous Integration** → frequent merges + automated testing = quick detection of problems.
- **TDD** → tests before code (developer perspective).
- **ATDD** → acceptance tests before code (business + user perspective).

What is White-Box Testing?

- **Definition:** Testing the *internal structure* or *logic* of the code.
- **Who does it?** Usually developers (sometimes testers with coding knowledge).
- **Goal:** Ensure every line, condition, and path in the code works correctly.
- **Opposite of Black-box testing:**
  - Black-box → "What does it do?" (functionality from outside)

- White-box → "How does it do it?" (logic inside the code)

Common White-Box Techniques

1. Statement Coverage

- Checks if **every line of code** executes at least once.
- *Example:*
- `if (x > 0) {`
- `    y = 1;   // statement 1`
- `}`
- `y = 2;       // statement 2`
  - If you test `x=5`, both statements run → 100% coverage.
  - If you only test `x=-1`, statement 1 is skipped → not full coverage.

Purpose: Make sure no line is left untested.

---

2. Branch Coverage (Decision Coverage)

- Ensures **every decision (true/false)** branch executes.
- *Example:*
- `if (x > 0) {`
- `    y = 1;   // branch A`
- `} else {`
- `    y = -1;  // branch B`
- `}`
  - If you test only `x=5`, you cover branch A but not branch B.
  - To get 100% branch coverage → test `x=5` and `x=-5`.

Purpose: Ensures all possible paths of decisions are tested.

---

3. Path Coverage

- Ensures **all possible paths** through the code are executed.
- Includes different combinations of branches.
- *Example:* Nested if-statements → multiple possible paths (T/T, T/F, F/T, F/F).

☞ **Purpose:** More thorough than branch coverage.

---

4. Condition Coverage

- Ensures each **boolean condition** inside decisions is tested both true and false.
- *Example:*
- `if (x > 0 && y > 0) {`
- `    z = 1;`
- `}`
    - You must test all combinations:
        - x>0, y>0
        - x>0, y<=0
        - x<=0, y>0
        - x<=0, y<=0

Purpose: Detects logic errors inside complex conditions.

Quick Comparison

Technique	What it ensures	Example requirement
Statement	Every line runs at least once	Run all statements
Branch	Each if/else decision takes both paths	True & False cases
Path	Every possible route through code covered	All combinations of branches
Condition	Every condition inside a decision is tested	For complex ifs

---

🔑 White Box Testing Techniques — Cheat Sheet

Technique	Definition (ISTQB)	What must be covered?	Typical Minimum Tests Needed	Example Code Snippet

Statement Coverage	Every **statement** must be executed at least once.	All executable statements.	Often **1** if chosen well.	if (x > 0) print("A");
				print("B");
				One test with x=1

Technique	Definition (ISTQB)	What must be covered?	Typical Minimum Tests Needed	Example Code Snippet
				covers both statements.
Decision/Branch Coverage	Every **decision outcome** (True & False) must be executed.	All branches of each decision.	**2 per decision** (at least).	`if (x > 0) print("A");` Need `x=1` (True) and `x=0` (False).
Condition Coverage	Each **atomic condition** in a decision must be True & False at least once.	All individual Boolean conditions.	≥ number of conditions × 2.	`if (A && B)` → Need A=T,F and B=T,F.
Decision + Condition Coverage	Combines Decision + Condition.	All decision outcomes **and** all conditions True/False.	More than branch coverage.	Same `if (A && B)` → Need cases for (T,T), (T,F), (F,T).
Multiple Condition Coverage	All possible **combinations** of atomic conditions.	Every condition combination.	**2^n** (n = number of conditions).	`if (A && B)` → 4 tests: (T,T), (T,F), (F,T), (F,F).
Path Coverage	Every **independent path** through the code must be executed.	All unique paths in control flow.	Potentially very high.	Nested ifs → exponential #paths.
LCSAJ Coverage	Covers Linear Code Sequence And Jump (start → end → jump).	Each sequence + its jump.	Depends on code structure.	Rarely in ISTQB Foundation, more advanced.
Data Flow Coverage	Based on variables: definition → use (def-use pairs).	All paths where variable values flow.	Depends on variables used.	`int x; x=1; y=x+2;` Must cover definition of `x` until its use.

🔥 **Quick "Minimum Tests" Shortcuts (ISTQB-style)**

- **Statement Coverage** → Often **1** test is enough.
- **Branch Coverage** → At least **2 per decision**.
- **Condition Coverage** → At least **2 × conditions**.
- **Multiple Condition Coverage** → **2^n** (n = conditions).
- **Path Coverage** → **All unique paths** (can explode quickly).

Remember for the exam:

- Statement = lines
- Branch = true/false of decisions
- Condition = each atomic Boolean
- Multiple condition = all combos
- Path = complete unique ways