🔹 **What is Object-Oriented Integration Testing?**

It's a way of testing **how objects in object-oriented programs work together** (integrate). We check if **classes and their methods** interact correctly.

🔹 **Why do we need it?**

In object-oriented programs, multiple classes (objects) **talk to each other**. Even if each class works fine on its own, they might **fail when combined**.

🔹 **Key Approaches (Keep these in mind with examples):**

1. **Top-Down Integration**
    - Test from **main (top) class** down to helper classes.
    - 🔸 *Example:* Start testing a ShoppingCart class, then test how it calls Product and PriceCalculator.
2. **Bottom-Up Integration**
    - Test **lower-level (independent)** classes first, then move up.
    - 🔸 *Example:* Test Product and PriceCalculator first, then ShoppingCart.
3. **Sandwich (Mixed) Approach**
    - Combines **top-down and bottom-up**.
    - 🔸 *Example:* Test ShoppingCart and Product at the same time, then move to middle components like DiscountManager.
4. **Big Bang Testing**
    - Combine all classes and test at once.
    - 🔸 *Example:* Plug all classes together (Cart, Product, Discount, User) and test the whole system.

⚠️ *Easy but risky* — hard to find where the bug is if something fails.

🔹 **Example to remember:**

Think of an **online shopping app**:

- User adds a Product to ShoppingCart
- ShoppingCart uses PriceCalculator
- We test **how all these objects interact** step-by-step or all together.

🔹 **What is Special Value Testing?**

It's a type of software testing where we check the program using **unusual or edge case values** — values that are **not normal or typical**, but still valid.

🔹 **Why use it?**

- To make sure the program works **correctly with special inputs**, like:
- Very **small** or **large numbers**
- **Zero**
- **Negative numbers**
- **Empty strings**
- **Null values**
- **Boundary limits** (like just before or after max/min)
- These values often cause **bugs**, so we test them carefully.

🔹 **Example: Login Form**

- Special values: empty string, null, very long strings
- You test:
- Username: "" → should reject
- Password: null → should reject
- Username: "aaaaaaaa...aaa" (1000 characters) → should be handled safely

🔹 **Tip to Remember:**

🧠 **"Test the weird stuff!"**
Special Value Testing = trying **unusual but valid** inputs to **catch hidden bugs**.

🔹 **What is Equivalence Testing?**

It's a testing method where inputs are **divided into groups (equivalence classes)** that should behave the same.
We **test one value from each group** instead of testing every possible input.

🟦 **Weak Normal vs Strong Normal Equivalence Testing**

Feature	**Weak Normal**	**Strong Normal**
What it tests	One value from **each valid class**, but **one at a time**	One value from **each valid class**, **combined in every possible way**
Number of inputs	One from each class separately	All **combinations** of inputs from all classes
Test cases	**Fewer** (faster but less thorough)	**More** (thorough but more effort)

Goal	Quickly check each valid input range	Check how **different valid inputs interact**
Example **2 Inputs**	A1 - B1	A1 – B1, A1 – B2, A2 – B1, A2 – B2

🔹 **What is Model-Based Testing?**

Model-Based Testing (MBT) is a technique where **you create a model (diagram or chart)** that shows how the software should behave — and then you generate test cases **from that model**.

🔹 **What is a "Model"?**

A **model** is like a **map or flowchart** that shows:

- **States** of the system (e.g. Login screen, Dashboard, Error page)
- **Transitions** (what happens when a user clicks a button, enters data, etc.)

📌 It's like a **blueprint** for how the software should behave.

🔹 **Why use MBT?**

- Saves time: Once the model is ready, test cases can be auto-generated
- Catches missing or incorrect behavior early
- Helps test **complex systems with many possible flows**

🔹 **Simple Example: ATM Machine**

Model States:

1. **Idle**
2. **Card Inserted**
3. **PIN Entered**
4. **Transaction Selected**
5. **Cash Dispensed**

Transitions:

- Insert card → Go to PIN screen
- Enter PIN → Go to menu
- Select "Withdraw" → Dispense cash

✅ From this **model**, we can create test cases like:

- Insert card → Enter correct PIN → Withdraw money
- Insert card → Enter wrong PIN → Show error

These test cases are **based on the model**, not random guessing.

🔹 **Tip to Remember:**

🧠 **"Model it, then test it!"**
Model-Based Testing = Use a diagram to plan test cases for **real-world behavior**

🔹 **Retrospective on MDD and TDD**

✅ **Model-Driven Development (MDD)**

What it is:
MDD focuses on **creating models (like UML diagrams)** first, and then generating code from those models.

Pros:

- Easy to visualize the system before coding
- Reduces coding errors by using formal models
- Helps in large and complex systems

Cons:

- Creating detailed models takes time and skill
- Not ideal for fast, small projects
- Generated code may need manual adjustments

Reflection:
MDD is useful when you need a **clear structure before coding**, especially for big projects. It helps teams **understand the system early**, but it can slow things down if models are too detailed or constantly changing.

✅ **Test-Driven Development (TDD)**

What it is:
TDD is where you **write tests before writing code**. You follow a cycle:

1. Write a failing test
2. Write code to pass the test
3. Refactor the code

Pros:

- Ensures your code works from the start
- Leads to cleaner, more reliable code
- Encourages small, focused functions

Cons:

- Slower at the beginning
- Can be hard when requirements are unclear
- Writing good tests requires experience

Reflection:
TDD helps create **bug-free and well-tested code**, especially in agile environments. It builds confidence, but it may slow early development and needs discipline to write good tests.

🔚 **Final Thoughts:**

- **MDD** is great for **planning and structure**
- **TDD** is best for **code quality and reliability**
- They can even be **combined**: use MDD to plan, and TDD to implement cleanly.

Here's a **super simple and easy-to-remember** explanation of **Boundary Value Testing (BVT)** with examples:

🔹 **What is Boundary Value Testing?**

Boundary Value Testing is a software testing technique where we test **values at the edges (boundaries)** of input ranges.

Why?
Because most bugs happen **at the boundaries**, not in the middle

🔹 **Why Use It?**

- Users often enter **min and max values**
- Developers often make mistakes near **limits**
- It helps catch errors like "off-by-one"

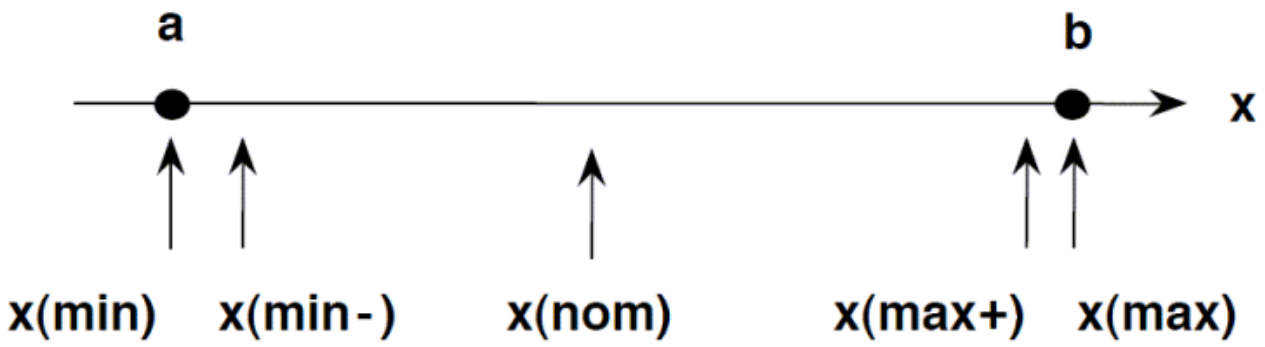
🔹 **How It Works**

For an input range **10 to 100**, you test:

- **Just below the boundary** → 9
- **At the lower boundary** → 10
- **Just above the lower boundary** → 11
- **Just below upper boundary** → 99
- **At the upper boundary** → 100
- **Just above upper boundary** → 101

✅ These 6 values help you catch most boundary-related bugs.

🔹 **Example: Age Input (Allowed: 18–60)**



- Test with:
    - 17 (just below – should fail)
    - 18 (boundary – should pass)
    - 19 (just above – should pass)
    - 59 (just below upper – should pass)
    - 60 (boundary – should pass)
    - 61 (just above – should fail)

🔹 **Tip to Remember:**

🧠 **"Most bugs live at the edges."**
So test just **before, at, and after** the **min and max** values.

🔹 **What is the Test-Then-Code Cycle?**

It's a **software development approach** where you **write tests before writing the actual code**.
This cycle is the heart of **Test-Driven Development (TDD)**.

🔁 **The Cycle Steps:**

1. **Write a Test**
    - Create a test case for a small function or feature.
    - At this point, the test will **fail** (because the feature doesn't exist yet).
2. **Run the Test**
    - Confirm that the test fails. This proves the test is working correctly.
3. **Write the Code**
    - Write the minimum amount of code to **make the test pass**.
4. **Run Tests Again**

- Check if all tests pass (new + old ones).

5. **Refactor the Code**
    - Clean up the code without changing its behavior. Make it neater or faster.
6. **Repeat** for the next feature or requirement.

🔹 **Simple Example:**

You want to write a function to add two numbers:

1. Write test: add(2, 3) should return 5
2. Test fails (no function yet)
3. Write def add(a, b): return a + b
4. Test passes
5. Clean up (if needed)
6. Done!

✅ **Benefits:**

- You always have **working, tested code**
- **Bugs are caught early**
- Code stays **clean and organized**

🧠 **Tip to Remember:**

Red → Green → Refactor
(Red = test fails, Green = test passes, Refactor = clean code)

This is the **Test-Then-Code Cycle** in action!

🔹 **What is SATM?**

SATM stands for **Software Automated Testing Methodology**.
It is a **structured approach** to designing, developing, and running **automated tests** in software testing.

🔹 **Purpose of SATM:**

- To improve **test efficiency**
- To ensure **repeatable, reliable, and fast testing**
- To reduce **manual testing effort**

🔹 **Key Components of SATM:**

1. **Test Planning**
    - Decide **what to automate**, which tools to use, and how the process will work

2. **Test Design**
    - Create **automated test cases** based on requirements or user stories
3. **Test Development**
    - Write the **scripts/code** for the automated tests
4. **Test Execution**
    - Run the automated tests on the software
5. **Test Reporting**
    - Collect and review **results** (passed, failed, errors)
6. **Maintenance**
    - Update tests if the software changes

🔹 **Example:**

Imagine testing a login form:

- SATM would guide you to:
    - Plan: Automate login test with Selenium
    - Design: Test cases like correct login, wrong password, blank fields
    - Develop: Write test scripts
    - Execute: Run the tests after every code change
    - Report: Get a summary of results
    - Maintain: Update tests if the login page changes

🧠 **Tip to Remember:**

SATM = A smart plan to build and manage automated tests
It's all about **making testing faster, repeatable, and reliable.**

🔹 **What is JUnit? And its features.**

JUnit is a unit testing framework for Java.

It allows developers to:

- Write tests for individual pieces of code (usually methods)
- Run them automatically
- Check if they pass or fail

JUnit is mostly used in Test-Driven Development (TDD).

🔹 **Key Features of JUnit:**

1. **Annotations**
    - Special keywords to control test flow
    - Examples:
        - @Test → Marks a method as a test
        - @BeforeEach → Runs before every test
        - @AfterEach → Runs after every test
2. **Assertions**
    - Used to check results
    - Example: assertEquals(5, add(2, 3))
      → Checks if result is 5
3. **Test Suites**
    - Group multiple test classes together to run as one
4. **Automatic Test Execution**
    - JUnit automatically runs all tests and shows:
        - ✅ Passed tests
        - ❌ Failed tests (with reasons)
5. **Integration with IDEs**
    - Works smoothly with Eclipse, IntelliJ, NetBeans, etc.
6. **Supports TDD**
    - Encourages writing tests before writing code
7. **Lightweight and Fast**
    - Easy to set up and run — good for continuous testing

🧠 Tip to Remember:

JUnit = Java Unit Testing
It helps you **test small parts of code**, quickly and automatically.

🔹 What is Slice-Based Testing?

Slice-Based Testing is a **white-box testing technique** that focuses on testing **data slices** of a program — specifically, **how data flows through code**.

It is based on the concept of **program slicing**, which means:

Extracting only the parts of the code that affect (or are affected by) a particular variable at a certain point.

🔹 **Why Use It?**

- Helps isolate and test **only the relevant parts** of code
- Useful for **debugging**, **understanding code**, and **creating efficient tests**
- Focuses on **data dependencies** and **control flow**

🔹 **Types of Slices:**

1. **Static Slice**
    - Based on **code structure**, doesn't depend on actual input values
    - Example: "Which lines of code affect variable x?"
2. **Dynamic Slice**
    - Based on a **specific input and execution path**
    - Example: "Which lines of code affected x during this run with input 5?"

🔹 **Example:**

```
int a = 5;
int b = 10;
int c = a + b;
System.out.println(c);
```

Let's say we're interested in the **value of c**.
A **slice** would include only the lines that **affect c**:

```
int a = 5;
int b = 10;
int c = a + b;
```

So instead of testing all code, we only test this "slice" — making testing **focused and efficient**.

🧠 **Tip to Remember:**

Slice-Based Testing = Test only the code that affects a specific variable.
It's like cutting out the **exact part** of the code that matters and testing just that.

🔹 **What is the Next Date Function?**

It's a function that **takes a valid date** (day, month, year) as input, and returns the **next calendar date**.

📅 Example:
Input → 28, 02, 2025
Output → 01, 03, 2025

🔹 **Why is it Complex?**

Because there are **many rules** to handle, like:

1. **Different days in months**
    - Jan: 31, Feb: 28/29, Apr: 30, etc.
2. **Leap years**
    - February has 29 days in leap years
3. **Month and year change**
    - 31 Dec → 01 Jan (next year)
4. **Input validation**
    - Dates like 31 April or 30 Feb are invalid

🔹 **Complexity Factors:**

Aspect	What Makes It Complex
Month handling	Different months have different max days
Leap year logic	Must check if year is divisible by 4, 100, 400
Boundary conditions	End of month / year changes need special care
Input validation	Invalid dates must be rejected
Day rollover logic	28 → 29 → 30 → 31 → 1 with month and year shift

🔹 **Example of Complexity:**

Input: 31, 12, 2024

- Day rolls over to 1
- Month rolls over to Jan
- Year increments to 2025

→ Output: 01, 01, 2025

All three fields (day, month, year) change — this is what makes the function **non-trivial** to test and code.

🧠 **Tip to Remember:**

Next Date Function Complexity = Handling calendar rules + boundary cases + leap years

That's why it's often used in **boundary value testing and decision table testing** examples.

🔹 **What is Structure-Based Testing?**

Structure-Based Testing is a software testing technique where **the tester uses knowledge of the internal code structure** to design test cases.

📌 It focuses on **how the program is built**, not just what it's supposed to do.

🔹 **Why Use It?**

- To make sure **every part of the code is executed at least once**
- To find **logical errors** or bugs **inside the code**
- To improve **code coverage** and **test reliability**

🔹 **Key Techniques in Structure-Based Testing:**

Technique	Description
Statement Testing	Test every line of code at least once
Branch Testing	Test every possible decision (if/else, switch cases)
Path Testing	Test all possible paths through the program
Condition Testing	Test each individual condition in decision-making
Loop Testing	Test loops with 0, 1, and many iterations

🔹 **Example:**

```
if (age >= 18) {
  System.out.println("Eligible to vote");
} else {
  System.out.println("Not eligible");
}
```

To fully test this, you'd use:

- age = 20 → covers the **true branch**
- age = 16 → covers the **false branch**

✅ This is **branch testing** — part of structure-based testing.

🔹 **Advantages:**

- Finds hidden logic bugs
- Ensures high code coverage
- Tests the actual **implementation**

🔹 **Disadvantages:**

- Needs access to source code

- Time-consuming for large systems
- Doesn't check **missing requirements**

🧠 **Tip to Remember:**

Structure-Based Testing = Test based on what's inside the code (structure, logic, flow)

It's like **looking inside the engine** instead of just testing the car from the outside.

✨ **Characteristics of Data Flow Testing (with Keywords):**

1. **Focus** – It focuses on **variables** and how they are defined, used, and killed (lifecycle tracking).
2. **Flow** – It uses a **control flow graph** to understand how data moves through the code.
3. **Error Detection** – Detects issues like:
    - **Undefined use** (using a variable before it's assigned),
    - **Unused definitions** (assigned but never used),
    - **Redefinition without use** (value overwritten before using).
4. **DU Chains** – Builds **Definition-Use (DU) chains** to check that every defined variable is followed by a proper use.
5. **Coverage** – Ensures **every variable definition** is tested through all its valid paths.
6. **Code Awareness** – It is a **white-box testing** technique, requiring knowledge of the program's internal logic.
7. **Clarity** – Helps improve **clarity and correctness** of how data is handled in loops and conditions.
8. **Quality** – Enhances overall **code quality** by verifying that variables are used efficiently and logically.

💡 **Tip to Remember**:
Think of **"Data Flow = Variable Journey"** — how data (variables) travel from definition to use across the code.

✅ **Top 10 Best Practices for Software Testing (with Keywords)**

1. **Plan** – *Start with a test plan*
   Always create a clear test strategy, schedule, and scope before testing.
2. **Understand** – *Know the requirements*
   Fully understand what the software should do before writing test cases.
3. **Prioritize** – *Test the most important first*
   Focus on **critical features** and **high-risk areas** first.
4. **Automate** – *Automate where possible*
   Use test automation for repetitive and regression tests to save time.

5. **Repeat** – *Regression testing is key*
   After every change or bug fix, **retest** to ensure nothing else broke.
6. **Explore** – *Use exploratory testing*
   Go beyond test cases to **manually explore** the app and find hidden bugs.
7. **Document** – *Keep records*
   Write clear test cases, bug reports, and results for **future reference**.
8. **Communicate** – *Talk to the team*
   Stay in sync with developers, analysts, and stakeholders for clarity.
9. **Review** – *Review everything*
   Regularly review test cases, test coverage, and results for improvement.
10. **Improve** – *Keep learning*
    Continuously learn new tools, techniques, and update your test process.

🎯 **Quick Mnemonic (first letters):**

PUPAREDCRI → *(Plan, Understand, Prioritize, Automate, Repeat, Explore, Document, Communicate, Review, Improve)*

---

2 Marks Questions

Define Software Testing:
Software testing is the process of checking if a software works correctly and finds bugs or errors.
👉 *Example:* Testing if a login button logs the user in properly.

Define Test Case:
A test case is a step-by-step instruction to test a specific part of the software. It includes input, expected output, and test steps.
👉 *Example:* Enter username & password → Click login → Expect dashboard to load.

Write a note on Equivalence Classes:
Equivalence classes divide input data into groups that should behave similarly, so we can test just one value from each group.
👉 *Example:* Age input: Valid = 18–60 → Pick 1 value like 30 to represent the group.

Mention any two issues of Object-Oriented Testing:

- Inheritance – One class can affect many others.
- Polymorphism – Same method behaves differently; hard to test.

What is Exploratory Testing?
Exploratory testing means testing without a script. Testers explore the application freely to find bugs using experience.
👉 *Example:* Clicking around a new app to see what breaks.

What is Fault and Error?

- **Fault (Bug)** – A defect in the code.
- **Error** – A **wrong action** by the user or system that leads to failure.
  👉 *Example:* Fault: wrong formula, Error: crash at runtime.

Boundary Value Testing:
Testing technique where we **focus on edge values** (like min, max) rather than typical values.
👉 *Example:* For input 1–100, test with 0, 1, 100, 101.

Guidelines for Equivalence Class Testing:

- Test one value from each class (valid/invalid).
- Include both **positive and negative** test cases.
- Don’t test multiple invalids together in weak class testing.

Random Testing:
Inputs are chosen **randomly**, not based on any logic or test design.
👉 *Example:* Auto-generated values used for form testing.

Difference between Weak Normal vs Strong Normal Equivalence Class Testing:

- **Weak Normal:** One value from each valid class, only **one combination at a time**.
- **Strong Normal: All possible combinations** of valid values are tested.

Slice-Based Testing:
A technique where we test **specific parts ("slices") of the program** based on variable usage.
👉 *Example:* Only test code lines that affect a specific variable's value.

💉 **1. White Box Testing**

Keyword: *Code-Aware Testing*

✅ **Definition:**

White Box Testing is a testing method where the **internal code, structure, and logic** of the software are tested.

🔑 **Key Points:**

- Tester knows the code.
- Focuses on how the system works internally.
- Used to check **paths, loops, conditions**, etc.
- Done mostly by **developers**.

💡 **Example:**

Testing each condition in an if-else block to ensure every path is covered.

🧪 **2. Black Box Testing**

Keyword: *Function-Focused Testing*

✅ **Definition:**

Black Box Testing is a method where the tester checks **what the system does**, **without knowing the internal code**.

🔑 **Key Points:**

- Tester does **not** need coding knowledge.
- Focuses on **input-output** behavior.
- Used to find **missing functions, UI errors, and performance issues**.
- Done mostly by **testers/QA team**.

💡 **Example:**

Entering a wrong password in a login form to see if the error message is shown — without knowing how the code handles it.
