Testing and Debugging

Dr. R. Mall

Software testing: Verification and validation,

Testing concepts-Failure, fault, test case, test suite and test script.

Levels of testing, Test plan, Test metrics and coverage, Role of testing-Testing strategies- -Black box and white box testing,

Unit tests-Integration testing-Top down integration-Bottom up integration-Validation testing-Alpha testing-Beta testing-

Other forms of high-level testing-Stress testing-Code inspections-

Manual testing-Automated testing-

Breaking tests-regression testing-Test case execution using testing frameworks

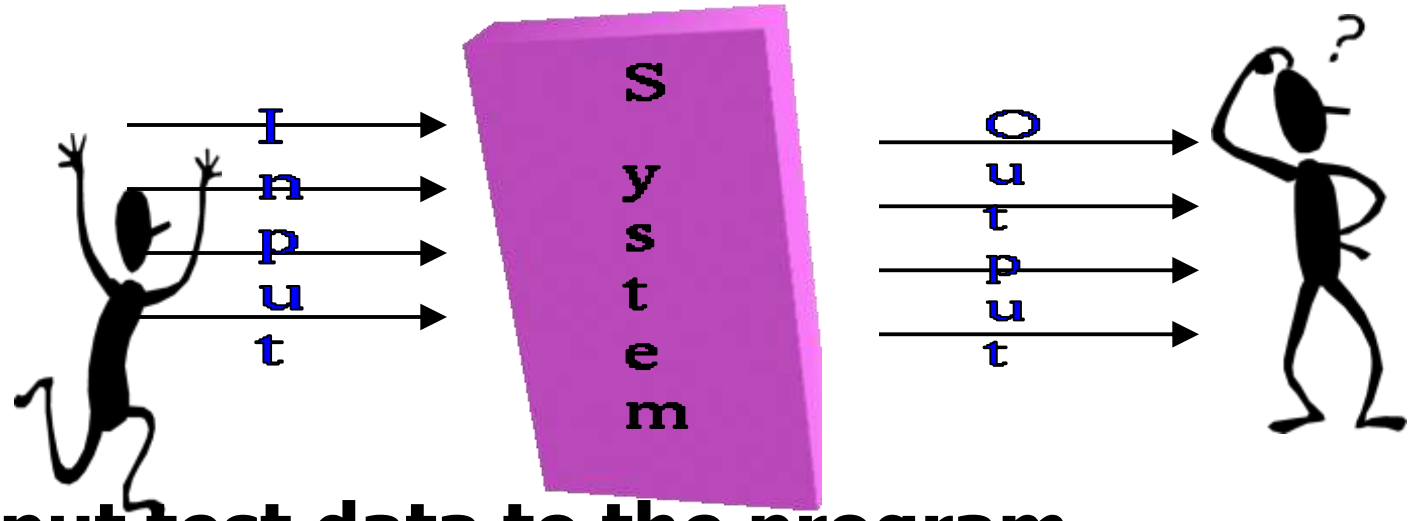
Examples of testing frame works (Tinderbox, JUnit, PyUnit)

and test automation tools(Selenium, Cucumber).

Organization of this lecture

- Important concepts in program testing
- Terminologies
- Level: Unit, Integration, and System testing
- Black-box testing: Types
- White-box testing: Types
- Summary

How do you test a system?



- Input test data to the program.
- Observe the output:
  - Check if the program behaved as expected.
  - If the program does not behave as expected:
    - note the conditions under which it failed.
    - later debug and correct.

Error, Faults, and Failures

- **Error->** programmer misunderstands the formula and writes area = 2 * π * r instead of π * r * r.

- mere presence of an error may not lead to a failure.
- **fault-> is an incorrect state entered during program execution: - it may or may not not lead to a failure.**
  - a variable value is different from what it should be.

The incorrect formula (area = 2 * π * r) is now part of the program's source code.-**That wrong line is the fault.**

Failure-The inability of the system to perform its intended function (deviation from expected behavior) during execution, caused by faults being triggered.

A failure is a manifestation of an error (aka defect or bug).

Levels of Testing

Level	Purpose	Example
Unit Testing	Test individual modules/functions.	Testing a login function separately.
Integration Testing	Check interaction between modules.	Login module with database connectivity.
System Testing	Verify the entire system against requirements.	Test the full banking app end-to-end.
Acceptance Testing	Validate system with user/business needs.	UAT: customer tests whether ATM works as expected.

Test cases and Test suites

- Test a software using a set of carefully designed test cases:
  - the set of all test cases is called the test suite .
  - A test case is a **triplet [I,S,O]**
    - **I is the data to be input to the system,**
    - **S is the state of the system at which the data will be input,**
    - **O is the expected output of the system.**

Test Plan

A formal document that outlines **what to test, how to test, when to test, and who will test**.

Typical contents:

- **Scope** – What is included/excluded in testing
- **Test items** – Features/modules to be tested
- **Test strategy** – Manual/automation, techniques used
- **Resources** – Tools, testers, environments
- **Schedule** – Timeline of testing phases
- **Exit criteria** – When to stop testing

Section	Details
Test Plan ID	ATM_TestPlan_001
System Under Test (SUT)	Automated Teller Machine (ATM) Software
Objective	Validate core ATM functionalities like card authentication, cash withdrawal, balance inquiry, and transaction safety.
Scope	- Functional Testing: Card insertion, PIN validation, cash withdrawal, receipt generation - Non-functional Testing: Response time, security checks
Out of Scope	- Hardware-level testing (cash dispenser motor) - Bank backend database validation
Test Items	Card reader module, PIN module, cash dispenser, transaction logs, UI messages
Features to be Tested	- Valid and invalid PIN entries - Cash withdrawal (sufficient & insufficient balance) - Transaction cancellation - Multiple failed login attempts (Card blocked)
Risks & Mitigation	- ATM hardware failure → Use simulator backup - Late code delivery → Prioritize critical test cases

Test Approach	- Unit testing on each module (PIN, Transaction) - Integration testing across modules - System testing for end-to-end flow - Regression testing after bug fixes
Test Environment	- OS: Windows/Linux server for backend - ATM simulator software - Database: Oracle
Test Data	- Valid card numbers & PINs - Expired/blocked cards - Accounts with low/high balance
Entry Criteria	- All code modules completed - Unit tests passed - Test environment ready
Test Deliverables	Test cases, test scripts, test data, defect reports, final test summary report
Roles & Responsibilities	- QA Engineer: Write and execute test cases - Test Lead: Approve test plan & track progress - Developer: Fix reported bugs
Schedule	- Test execution: 15th – 25th Sept - Regression testing: 26th – 28th Sept

Test Script

Depending on context, a test script can be:

1. **Manual Test Script** – A written step-by-step guide for a tester to manually execute, such as:
   Step 1: Open the login page.
   Step 2: Enter a valid username and password.
   Step 3: Click "Login."

   Expected Result: The user is redirected to the dashboard.

2. **Automated Test Script** – A piece of code (written in tools like Selenium, JUnit, PyTest, etc.) that automatically performs the same steps and checks the results without human intervention.

A test case is the idea of what to test, while a test script is the actual detailed instructions (manual or automated) to perform the test.

```
# Import Selenium WebDriver
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

# Create a WebDriver instance (using Chrome)
driver = webdriver.Chrome()

# Step 1: Open the login page
driver.get("https://example.com/login")

# Step 2: Enter username
username_box = driver.find_element(By.NAME, "username")
username_box.send_keys("test_user")

# Step 3: Enter password
password_box = driver.find_element(By.NAME, "password")
password_box.send_keys("secure_password")

# Step 4: Click login button
login_button = driver.find_element(By.ID, "loginBtn")
login_button.click()

# Step 5: Check expected result (dashboard appears)
expected_url = "https://example.com/dashboard"
if driver.current_url == expected_url:
    print("Test Passed: Login successful")
else:
    print("Test Failed: Login did not redirect correctly")

# Close browser
driver.quit()
```

- **Precondition: The login page must be accessible.**
- **Steps: Open page → enter credentials → click login.**
- **Expected Result: The browser should redirect to the dashboard.**
- **Actual Result: Script checks the URL and reports pass/fail.**

This script can be reused and expanded (e.g., invalid login test, empty fields, etc.).

Why Testing- Role of Testing

- **Validation** – Ensure system meets customer requirements.
- **Verification** – Check system is built correctly according to design.
- **Quality Assurance** – Improves reliability, performance, and security.
- **Risk Reduction** – Finds bugs early, saves cost and time.
- **Customer Confidence** – Ensures product works as promised.

Verification vs Validation

- Verification is the process of determining:
  - whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining
  - whether a fully developed system conforms to its SRS document.
- Verification is concerned with **phase containment of errors**,
  - whereas the aim of validation is that the **final product be error free**.

Terminologies

Levels = *where testing happens (unit → system → acceptance)*

Test plan = *roadmap of testing*

Metrics = *how we measure testing quality*

Role = *why testing matters (quality, confidence, risk reduction)*

Design of Test Cases

A test case is a set of conditions, inputs, execution steps, and expected results created to verify that a software application works as intended.

Steps in Test Case Design

1. **Understand Requirements** → Extract functional and non-functional requirements from SRS.
2. **Identify Test Scenarios** → Determine what features/flows need testing.
3. **Define Inputs & Preconditions** → Data and system state before executing the test.
4. **Determine Expected Results** → Based on requirements.
5. **Design Test Cases** → Write detailed steps with clear input/output.
6. **Review & Optimize** → Remove duplicates, ensure coverage.

Example of Test Cases: ATM Withdrawal Function

Requirement: A user can withdraw cash if balance ≥ withdrawal amount, within daily limit.

Test Case ID	Description	Preconditions	Input	Steps	Expected Result
TC001	Valid withdrawal	User has ₹5000	Withdraw ₹2000	Enter PIN, request withdrawal	Amount dispensed, balance reduced
TC002	Exceed balance	User has ₹2000	Withdraw ₹5000	Enter PIN, request withdrawal	Transaction denied, “Insufficient Balance” message
TC003	Exceed daily limit	Limit = ₹10,000	Withdraw ₹12,000	Enter PIN, request withdrawal	Transaction denied, “Daily Limit Exceeded” message
TC004	Invalid PIN	User has ₹5000	Enter wrong PIN 3 times	Attempt withdrawal	Account locked after 3 attempts

Test Case DesignTechniques

- **Black Box Techniques** (focus on input-output, without code knowledge):

1.**Equivalence Partitioning (EP):** Divide input data into valid/invalid classes.
2.**Boundary Value Analysis (BVA):** Test at edges (e.g., min, max, just inside/outside values).
3.**Decision Table Testing:** Test combinations of inputs and their effects.
4. **State Transition Testing:** Verify system behavior under different states.

White Box Techniques (need knowledge of code structure):

Statement Coverage
Branch Coverage
Path Coverage

Experience-Based Techniques:

- **Error guessing (tester's intuition).**
- **Exploratory testing.**

Blackbox/functional testing.

- Test cases are designed using only **functional specification** of the software:
- **without any knowledge of the internal structure of the software.**
- For this reason, black-box testing is also known as functional testing.

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
  - program behaves in similar ways to every input value belonging to an equivalence class.

Equivalence Class Partitioning

- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

Why define equivalence classes?

- Test the code with just one representative value from each equivalence class:
  - as good as testing using any other values from the equivalence classes.

Equivalence Partitioning (EP): Divide input data into valid/invalid classes.

- Input data is divided into **partitions (classes)** where system behavior is expected to be the same.
- **Instead of testing all values, you test one from each partition.**

Case Study: Online Shopping – Age-based Registration

- **Requirement: A user must be 18–60 years to register.**

Equivalence Classes:

- **Valid class:** $18 \leq Age \leq 60$
- **Invalid class 1:** $Age < 18$
- **Invalid class 2:** $Age > 60$

Partition Type	Input Example	Expected Result
Invalid Class 1 (Age < 18)	17	Registration Denied
Valid Class (18 ≤ Age ≤ 60)	25	Registration Allowed
Invalid Class 2 (Age > 60)	65	Registration Denied

Boundary Value Analysis (BVA) – ATM Withdrawal Limit (₹100 – ₹10,000)

- **Errors often occur at boundaries of input domains.**
- **Test minimum, maximum, just inside, just outside values.**
- **Boundary Values:**
- **Minimum = 100**
- **Maximum = 10,000**

Test Condition	Input Amount	Expected Result
Just below min	₹99	Denied
Minimum value	₹100	Allowed
Just above min	₹101	Allowed
Just below max	₹9,999	Allowed
Maximum value	₹10,000	Allowed
Just above max	₹10,001	Denied

Decision Table Testing-

Used when multiple conditions/inputs interact and produce different outcomes.

- It ensures all combinations of inputs are covered.

Case Study Example: Online Loan Approval System

Inputs:

1. Applicant has minimum salary? (Yes/No)
2. Applicant has good credit score? (Yes/No)
3. Applicant has no pending loans? (Yes/No

Case	Salary ≥ 30k	Good Credit Score	No Pending Loans	Decision
1	Y	Y	Y	Approved
2	Y	Y	N	Rejected
3	Y	N	Y	Rejected
4	N	Y	Y	Rejected

State Transition Testing

Used when the system behavior depends on the current state and an event that triggers a transition.

Case Study : ATM Machine: Transition Table

Current State	Event / Input	Next State	Expected Output / Behavior
Idle	Insert Card	Card Inserted	Prompt for PIN
Card Inserted	Remove Card (before PIN)	Idle	Card ejected, system resets to Idle
Card Inserted	Enter wrong PIN (3 times)	Card Blocked	Retain card, display “Card Blocked”
Card Inserted	Enter correct PIN	Authenticated	Access granted, menu displayed
Authenticated	Select Withdraw + Confirm	Transaction	Dispense cash, update balance
Transaction	Take Cash	Idle	Return to Idle state

Use Case for BLACKBOX TESTING : ATM Cash Withdrawal System

Description:

A user inserts a card, enters a PIN, and requests an amount `A` to withdraw.

- **Constraints:**

  `0 < A ≤ account_balance`

  `A ≤ daily_limit`

  `A` **must be in multiples of 100**

- **Output:**

  - **Dispense cash (if valid)**
  - **Error message (if invalid)**

- **1. Equivalence Partitioning (EP)-Divide inputs into valid/invalid partitions.**

- **Valid Partitions**
  - `A > 0`, `A ≤ balance`, `A ≤ daily_limit`, `A % 100 = 0` → **Success**
- **Invalid Partitions**
  - `A = 0`
  - `A < 0`
  - `A > balance`
  - `A > daily_limit`
  - `A % 100 ≠ 0`

- **Constraints:**
  `0 < A ≤ account_balance`
  `A ≤ daily_limit`
  `A` **must be in multiples of 100**

- 👉 **Example Tests:**
- `A=500` **(valid if balance=1000, limit=2000) → Success**
- `A=250` **(not multiple of 100) → Error**
- `A=3000` **(exceeds daily limit) → Error**

2. Boundary Value Analysis (BVA)

Check values at edges of valid/invalid partitions.

Assume: `balance = 1000`, `daily_limit = 2000`.

- **Minimum valid: `A = 100`**
- **Just below min: `A = 0` (invalid)**
- **Just above min: `A = 200`**
- **Just below max: `A = 2000` (valid)**
- **Max limit: `A = 2000`**
- **Just above max: `A = 2001` (invalid)**

- **Constraints:**
  `0 < A ≤ account_balance`
  `A ≤ daily_limit`
  `A` **must be in multiples of 100**

👉 **Example Tests: `A={0,100,200,1999,2000,2001}`**

Decision Table Testing

Conditions involve **balance, daily limit, and multiple of 100**.

Condition	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
`A ≤ balance`	Y	Y	N	Y	Y
`A ≤ daily_limit`	Y	N	Y	Y	Y
`A % 100 == 0`	Y	Y	Y	N	Y
Action	Success	Error	Error	Error	Error

Example Tests:

- `A=500` → **Rule 1 → Success**
- `A=3000` → **Rule 2 → Error (exceeds daily limit)**
- `A=250` → **Rule 4 → Error (not multiple of 100)**

State Transition Testing

ATM can be modeled as a state machine:

- **States**
  - `Idle` **(waiting for card)**
  - `Card Inserted` **(waiting for PIN)**
  - `Authenticated` **(waiting for amount input)**
  - `Dispense Cash`
  - `Error`
- **Transitions**
  - `Idle → Card Inserted` **(card entered)**
  - `Card Inserted → Authenticated` **(valid PIN)**
  - `Authenticated → Dispense Cash` **(valid request)**
  - `Authenticated → Error` **(invalid request)**
  - `Error → Idle` **(reset transaction)**

Example Test

- **Insert card → Enter PIN → Request 500 (valid) → Cash Dispensed**
- **Insert card → Enter PIN → Request 250 (invalid multiple) → Error**

White-box/Glass box testing/ Structural testing

involves testing the **internal logic, control flows, and structure of the program.**

The tester must have **knowledge of the source code.**

There exist several popular white-box testing methodologies:

- **Statement coverage**
- **branch coverage**
- **path coverage**
- **condition coverage**

1. Statement Coverage

Ensures that every statement in the program is executed at least once during testing.

Example: If code has if-else, both blocks should be executed in some test case.

Formula:

$$\text{Statement Coverage} = \frac{\text{Number of statements executed}}{\text{Total number of statements}} \times 100$$

2. Branch Testing

- Ensures that all **decision outcomes (True/False branches) are executed at least once.**

- Example: **For if(condition), both the True and False outcomes must be tested.**
  Formula:

$$\text{Branch Coverage} = \frac{\text{Number of branches executed}}{\text{Total number of branches}} \times 100$$

3. Condition Coverage-More thorough than just branch testing.

- Ensures that each **boolean sub-expression** in a decision is evaluated to both True and False.
- Example: For `if(A && B)`, tests should make A=True/False and

1. Example: Condition Coverage

Code Snippet:

```
if (A > 5 and B < 10):
    print("Condition True")
else
    print("Condition False")
```

- **Here, the decision is `(A > 5 and B < 10)`**

  It has two sub-conditions:

  `A > 5 and B < 10`

Test Case	A	B	A > 5	B < 10	Final Result
1	6	9	True	True	True
2	6	15	True	False	False
3	2	8	False	True	False
4	2	15	False	False	False

Cyclomatic Complexity: Pre-requisite for Path Coverage

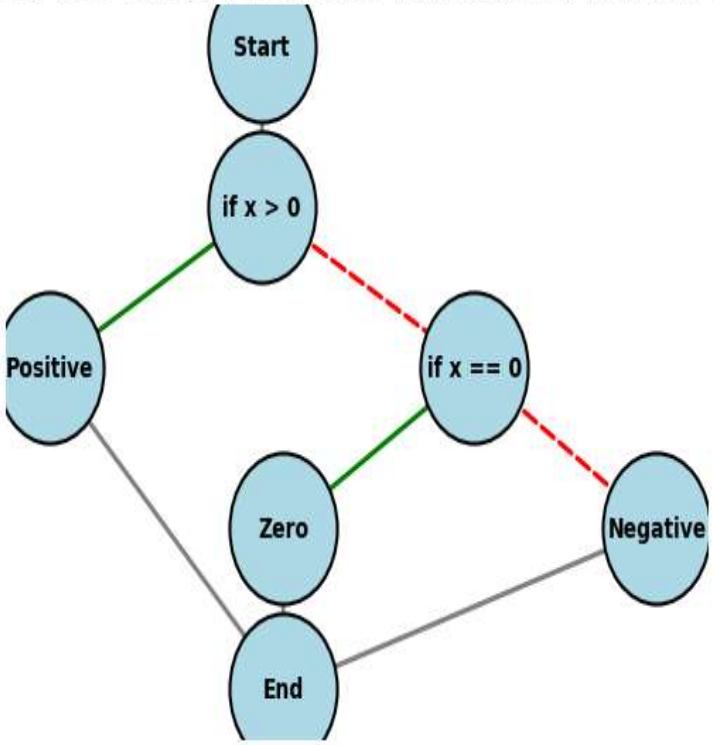
- A metric to measure the **complexity of a program's control flow**.
- It helps to determine the **minimum number of test cases required** for complete branch/path coverage.

- **Formula (McCabe's): V(G)=E−N+2P** where:
  E = number of edges in control flow graph
  N = number of nodes
  P = number of connected components (usually 1 for a single program)

- **Higher cyclomatic complexity ⇒ more test cases needed ⇒ code is harder to test and maintain.**
- **Cyclomatic Complexity → Numeric value that tells how many independent paths need to be tested .**

```
def check_number(x):
    if x > 0:
        print("Positive")
    else:
        if x == 0:
            print("Zero")
        else:
            print("Negative")
```

Control Flow Graph (CFG) with Highlighted Negative Arcs



Step 1: Build Control Flow Graph

Nodes:
1 = start,
2 = if x > 0,
3 = print("Positive"),
4 = if x == 0,
5 = print("Zero"),
6 =print("Negative"),
7 = end
Edges: connect nodes according to control flow.

$$V(G) = E - N + 2P$$

- E (Edges) = 8
- N (Nodes) = 7
- P (Connected components) = 1

So,

Step 2: Apply McCabe's Formula

$$V(G) = 8 - 7 + 2(1) = 3$$

Control Flow Graph for Path Coverage Testing

- **Sequence in which different instructions of a program get executed.**
- the **way control flows** through the program.
- **Number all the statements of a program.**
  - **they represent nodes of the control flow graph.**
- **An edge from one node to another node exists:**
  - **if execution of the statement representing the first node**
  - **can result in transfer of control to the other node.**

Step 3: Interpretation

Cyclomatic Complexity = 3 → Minimum **3 independent test cases** are required to cover all paths.

Test Cases Needed:->ensures **complete path/branch coverage**.

- `x > 0` → covers Positive branch
- `x == 0` → covers Zero branch
- `x < 0` → covers Negative branch

Thus **McCabe's metric provides:**

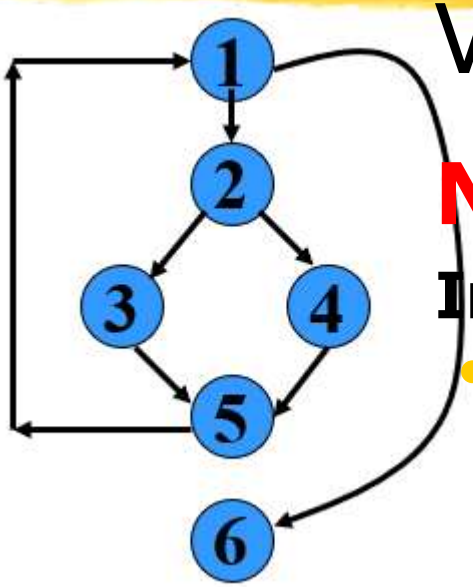
quantitative measure of testing difficulty and the ultimate reliability

The **cyclomatic complexity of a program** provides:

- **a lower bound on the number of test cases to be designed**
- **to guarantee coverage of all linearly independent paths.**

Bounded area

- Any region enclosed by a nodes and edge sequence.



Visual examination of the CFG:

No of bounded areas is 2.

Intuitively,

- **number of bounded areas increases with the number of decision nodes and loops.**

Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

69

Knowing the number of test cases required:

does not make it any easier to derive the test cases,

only gives an indication of the minimum number of test cases required.

4. Path Coverage

- Ensures that **all possible paths** in the program's control flow are executed.

  Stronger than statement and branch coverage.
- **Example: In nested if-else structures, every possible route must be tested.**
- A **dynamic program analyzer** is used:
  a. to **indicate which parts of the program have been tested**
  b. the output of the dynamic analysis is used to guide the tester in selecting additional test cases.

```
def check_number(x):
    if x > 0:
        if x % 2 == 0:
            return "Positive Even"
        else:
            return "Positive Odd"
    else:
        return "Non-Positive"
```

Step 1: Identify Paths

Control flow decisions:

1. `if x > 0` → True / False
2. `if x % 2 == 0` → True / False

Possible execution paths:[So there are 3 distinct paths.]

1. `x > 0` ->True → `x % 2 == 0` True → "Positive Even"
2. `x > 0` ->True → `x % 2 == 0` False → "Positive Odd"
3. `x > 0` False → "Non-Positive"

Step 2: Test Cases for Path Coverage

- Path 1: `check_number(4)` → Positive Even
- Path 2: `check_number(3)` → Positive Odd
- Path 3: `check_number(-2)` → Non-Positive

Step 3: Path Coverage Achieved All **execution paths** are tested.

This ensures maximum confidence in the control flow correctness.

Summary:

- Path coverage tests **all possible routes** through the program.
- Example above required **3 test cases** for full coverage.

use **Euclid's GCD algorithm** as a case study and see how different **white-box testing strategies** apply.

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

1. Statement Coverage

Ensure every statement in the program is executed at least once.

- **Need test cases where both the `if a > b` and `else` blocks are executed.**

Example Test Cases:

- `gcd(10, 6)` → **executes** `a > b` **branch**
- `gcd(6, 10)` → **executes** `else` **branch**

This covers all statements.

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

2. Branch Coverage

Ensure all decision outcomes (True/False) are tested.

Decisions:

1. `while a != b` → **True and False**
2. `if a > b` → **True and False**

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Example Test Cases:

- `gcd(10, 6)` → `if a > b = True`
- `gcd(6, 10)` → `if a > b = False`
- `gcd(8, 8)` → `while a != b = False` **immediately (loop skipped)**

  This ensures both branches of **while** and **if** are covered.

3. Condition Coverage

Ensure that each **boolean condition** is tested for both True and False.

Conditions here:

- `a != b`
- `a > b`

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Example Test Cases:

- `gcd(10, 6)` → `a != b = True`, `a > b = True`
- `gcd(6, 10)` → `a != b = True`, `a > b = False`
- `gcd(8, 8)` → `a != b = False`

Both conditions are evaluated as **True and False**.

4. Path Coverage

Ensure **all possible execution paths** are tested.

Possible paths:

1. **Direct return (when `a == b`)**
2. **Loop with `if a > b` (subtraction of `a`)**
3. **Loop with `else` (subtraction of `b`)**

Example Test Cases:

- `gcd(8, 8)` → **Path 1**
- `gcd(10, 6)` → **Path 2**
- `gcd(6, 10)` → **Path 3**

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Cyclomatic Complexity= 3

Thus min of 3 test cases needed

We measure the number of independent paths.

Control flow decisions:(there are 2 decisions as below:)

- `while a != b` **(1 decision)**

- `if a > b` **(2**

$$V(G) = E - N + 2P$$

$$V(G) = \text{Number of decisions} + 1$$

Strategy	Goal	Example Test Cases
Statement Coverage	Execute all statements at least once	(10,6), (6,10)
Branch Coverage	Cover True/False of all decisions	(10,6), (6,10), (8,8)
Condition Coverage	Each condition tested True & False	(10,6), (6,10), (8,8)
Path Coverage	Cover all unique execution paths	(8,8), (10,6), (6,10)
Cyclomatic Complexity	Independent paths in program (min tests)	Value = 3

1. Mutation Testing

```
def is_even(x):
    return x % 2 == 0
```

fault-based testing technique where small, artificial changes (mutations) are made in the program code to create **mutant versions**.

The goal is to check whether the existing test cases can **detect these changes**.

- If the test case fails for the mutant → the mutant is **killed**.
- If the test case passes (i.e., does not detect the change) → the mutant **survives** (test suite is weak).

Purpose: To measure the **effectiveness of test cases**.

Mutation Testing

```
def is_even(x):
    return x % 2 == 0
```

Mutation testing ensures

hig

1. Change operator `%` to `/` → `return x / 2 == 0`
2. Change `==` to `!=` → `return x % 2 != 0`
3. Change constant `0` to `1` → `return x % 2 == 1`

Po

Test Cases:

- **Input: x = 4 → Expected: True**
- **Input: x = 5 → Expected: False**

- **With mutant #2 (!=),**

test with x=5 → returns True (wrong) → mutant killed.

If no test kills a mutant → we need better test cases.

Data Flow–based Testing

```
def compute_sum(x, y):
    s = x + y      # Definition of s
    if s > 10:     # Use of s
        return "Large"
    else:
        return "Small"
```

Data flow–based testing focuses on how data (variables) are defined, used, and killed (lifetime).

The idea is to design test cases that cover all important **definition–use (DU) paths** of variables

- **Def**: Where a variable is **assigned a value**
- **Use**: Where a variable is **read/used**.
- **Kill**: Where a variable is no longer needed (goes out of scope).
- **Variable s is defined at s = x + y.**
  s is used in condition if s > 10.
- **These test cases in green box**

  ensure the variable s's definition

  and usage are both exercised.

Test Cases for Data Flow Coverage

- **Case 1:**

`compute_sum(3, 4)` → `s = 7`, path covers **def** → **use** → goes to `"Small"`.

- **Case 2:**

`compute_sum(8, 5)` → `s = 13`,

path covers def → use → goes to `"Large"`.

Aspect	Mutation Testing	Data Flow Testing
Focus	Effectiveness of test suite (fault injection)	Variable definitions & usages
Method	Modify code (mutants) and re-run tests	Analyze variable DU paths
Goal	Kill all mutants with strong test cases	Ensure all data flows are covered
Example	Change `==` to `!=` and see if test catches	Check `s = x+y` is used in condition

Summary:Black-box Testing

- Test cases are designed using only functional specification of the software:
  - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

Summary: White-box Testing

- Designing white-box test cases:
  - requires knowledge about the internal structure of software.
  - white-box testing is also called structural testing.

Testing Type	Description	Example Scenario
Unit Testing	Tests individual modules or functions in isolation.	Checking if the login function returns "valid user" for correct credentials.
Integration Testing	Ensures different modules interact correctly.	Payment gateway + Order module in Swiggy.
Top-down Integration	Integration begins from the top-level module and goes downward, stubs used for lower levels.	Testing ATM UI first with dummy backend modules.
Bottom-up Integration	Integration begins from lower-level modules, drivers used to simulate upper layers.	Testing database functions before UI in an ERP.
Validation Testing	Ensures the system meets user needs and requirements.	Checking if "withdraw cash" works as per ATM requirement.
Alpha Testing	Internal testing by developers/QA within the organization.	Company employees testing a new app before release.
Beta Testing	External testing by limited end-users before official launch.	Selected users trying WhatsApp beta version.
Test Case Execution using Frameworks	Using test frameworks (JUnit, PyTest, TestNG) to execute structured test cases.	Running JUnit tests on Java banking software.

Stress Testing	**Puts system under extreme conditions to test limits.**	**Simulating 10,000 users on a food delivery app at once.**
Code Inspections	**Manual review of source code for defects.**	**Team members reviewing login module code for errors.**
Manual Testing	**Human testers execute test cases without automation tools.**	**QA tester clicking through UI screens to check flow.**
Automated Testing	**Tools/frameworks execute test scripts automatically.**	**Using Selenium to test web form validation.**
Breaking Tests	**Intentional attempts to break the system by unusual inputs.**	**Entering "9999999999" as age in a signup form.**
Regression Testing	**Re-testing after code changes to ensure old features still work.**	**After adding "discount coupons", check if ordering still works.**

A **testing framework** is a set of guidelines, rules, or libraries that provide structure for writing and executing test cases. It generally supports **unit testing**, **integration testing**, or **system testing**.

Framework	Language/Platform	Description
JUnit	Java	Popular unit testing framework for Java applications; supports annotations, assertions, and test runners.
PyUnit (unittest)	Python	Built-in unit testing framework in Python; similar to JUnit; supports test suites, fixtures, and assertions.
Tinderbox	Various	Early automated testing framework (often historical reference; originally from Mozilla) that builds and runs tests automatically.
TestNG	Java	Inspired by JUnit, supports parallel testing, annotations, and flexible configuration.
NUnit	.NET	Unit testing framework for .NET languages; similar to JUnit but for C# or VB.NET.

Summary

- If we select test cases randomly:
  - many of the test cases may not add to the significance of the test suite.
- There are two approaches to testing:
  - black-box testing
  - white-box testing.

Summary

- Black box testing is also known as functional testing.
- Designing black box test cases:
  - requires understanding only SRS document
  - does not require any knowledge about design and code.
- Designing white box testing requires knowledge about design and code.

Summary

- We discussed black-box test case design strategies:
  - equivalence partitioning
  - boundary value analysis
- We discussed some important issues in integration and system testing.