# Software Engineering (Code: CSE3122)

B. Tech CSE 5th Sem.

(School of Computer Engineering)

**Dr. Kranthi Kumar Lella**
**Associate Professor,**
**School of Computer Engineering,**
**Manipal Institute of Technology,**
**MAHE, Manipal – 576104.**

# SOFTWARE ENGINEERING

## Module – 7
## (CODING AND TESTING)

Coding, Code review, Software Documentation, Testing, Unit Testing, Black-Box testing, White-Box Testing, Debugging, Program Analysis tools, Regression testing, Security testing, Robustness testing, Fuzzy testing, Integration testing, Testing OOP, System testing, Some general issues associated with testing.

# 🔷 Coding and Testing: Overview

This Module focuses on the **implementation and verification** stages of software development, specifically:

- ➢ **Coding Phase**
- ➢ **Testing Phase**

## ✅ Coding Phase

### 📌 What Happens:

- Each **module** designed during the design phase is **coded (implemented)**.
- Once coded, each module undergoes **unit testing** — tested **in isolation** to verify its correctness.

### 🔧 Key Activities:

- Writing code based on the **design document**.
- **Unit testing** modules as soon as they're coded.
- Ensuring each module functions **independently** before integration.

## 🔗 Integration and Testing

### 📌 Integration Plan:

- Modules are **integrated step by step** following a predefined **integration plan**.
- In each step, new modules are added to the **partially integrated system**, and the **combined system is tested**.
- Gradually, the **complete system** is built and tested.

## System Testing

### 📌 What It Involves:

- Done on the **fully integrated product**.
- Software is tested **against the Software Requirements Specification (SRS)** to ensure it behaves as expected.

## 🔎 Importance of Testing

### ⚠️ Reality Check:

- **Testing is the most effort-intensive phase** in software development.
- A **large number of test cases** are required to validate functionality, performance, and robustness.
- **Parallel execution of test cases** is common to save time.

### 👨‍💻 Manpower:

- Testing teams are often **larger than design/coding teams**.
- There's a **high demand** for **software test engineers** in the industry.

### ❌ Misconception:

- Some beginners think testing is **less intellectual** than design or coding.
- However, **modern testing** requires strong understanding of:
  - Testing **techniques**
  - **Tools**
  - **Creative thinking** to uncover bugs

# 7.1 CODING

## 🎯 Main Objective of the Coding Phase

To **transform the design document** into working **code in a high-level programming language**, followed by **unit testing** of each module.

## 🧾 Input to the Coding Phase

The **Design Document**, which contains:

- **High-level design** (structure charts showing module call relationships)
- **Detailed design** (module specifications including algorithms and data structures)

**Example:** Suppose your design includes ***calculateBill()*** and ***applyDiscount()***. Each must now be coded separately and tested individually.

## 💼 What Happens in the Coding Phase?

- Every module identified in design is **coded** as per its **specifications**.
- **Unit testing** is done immediately after coding each module.
- **Code reviews** are conducted to:
  - Check for **errors**
  - Ensure **coding standard compliance**

## 🔒 Coding Standards vs Coding Guidelines

| Feature | Coding Standards | Coding Guidelines |
|---|---|---|
| Meaning | Rigid, formal rules that must be followed | Suggestions or good practices |
| Enforceability | Mandatory – checked during code review | Optional – developer discretion |
| Purpose | Ensure uniformity and consistency | Encourage clarity, maintainability, and efficiency |
| Example | Variable naming conventions, error codes | avoid cleavage coding, Don't re-use variables |

## 📌 Benefits of Coding Standards

1. Uniform appearance across the codebase
2. Easier understanding, debugging, and maintenance
3. Promotes **good programming practices**
4. Supports **code reuse**

## Example Coding Standards

## Variable Naming

- **Global variables**: Start with uppercase (e.g., GlobalCount)
- **Local variables**: Start with lowercase (e.g., tempValue)
- **Constants**: All uppercase (e.g., MAX_SIZE)

## 📄 Module Header Format

Should include:

- Module Name
- Creation Date
- Author Name
- Modification History
- Synopsis (short description)
- Functions list with I/O
- Global variables used

## 🚨 Error and Exception Handling

- Consistent return values for errors (e.g., 0 for success, 1 for failure)

## 📘 Example Coding Guidelines (Best Practices)

| Guideline | Reason |
|---|---|
| ❌ Don't write clever or cryptic code (undesirable splitting or breaking up of code logic) | Makes code hard to understand and maintain |
| ❌ Avoid uncertain side effects | E.g., hidden global variable changes or I/O |
| ❌ Don't use one variable for multiple purposes | Leads to confusion and bugs during enhancement |
| ✅ Use meaningful variable names | Enhances code readability |
| ✅ Add adequate comments | Rule of thumb: 1 comment per 3 lines of code |
| ❌ Avoid GO TO statements | Makes code unstructured and hard to debug |
| ❌ Limit function size to 10 lines max | Keeps functions simple, modular, and testable |

## 📌 Code Review

- ❖ Ensures compliance with coding standards
- ❖ Identifies errors early (before testing)
- ❖ **Mandatory** in good software development practices
- ❖ Code that **violates standards** is **rejected** and reworked

# ✅ 7.2 CODE REVIEW

## 🎯 Objective

To detect **logical, algorithmic, and programming errors** in the code **before testing** begins.

It is **more cost-effective** than testing because:

- Reviews **directly detect errors**, skipping the lengthy debugging process.
- Debugging after testing is **labor-intensive** and time-consuming.

## 📌 When is Code Review Performed?

- **After successful compilation** of a module (i.e., all syntax errors are fixed).
- Targets **semantic** and **logical** errors, not syntax issues.

## Benefits of Code Review

- Reduces cost of error detection.
- Produces **high-quality, maintainable code**.
- Helps teams identify recurring coding issues and refine practices.
- Improves **coding style**, **algorithm selection**, and **design feedback**.

## 🔍 Types of Code Review

### 1. Code Walkthrough

An **informal** review technique.

**How it works:**

- Small group of developers (ideally **3 to 7 members**).
- Members review code a few days before the **walkthrough meeting**.
- Each simulates execution of the code **by hand** using **selected test cases**.
- Meeting is held to **discuss findings**.

### ✅ Guidelines for Effective Walkthroughs:

- Focus on **error discovery**, not **error fixing**.
- **Managers should not attend**, to encourage open, non-judgmental discussion.
- Avoid large teams; keep it small and focused.

### 🎯 Objective:

To uncover **algorithmic and logical flaws** through **mental simulation**.

## 2. Code Inspection

A **formal review** technique aimed at detecting:

- Common coding errors
- Violations of **coding standards**
- Programming oversights

### 🧰 Inspection Checklist (Sample Errors):

- Use of **uninitialized variables**
- **Infinite loops** or incorrect loop conditions
- **Array out-of-bounds** access
- **Incorrect parameter** passing
- **Improper memory** allocation/deallocation
- Use of **wrong logical operators** or precedence
- **Dangling references** (use of unallocated memory)
- **Floating-point comparison** issues

### 📈 Benefits Beyond Bug Detection:

- Educates developers through **exposure to others' mistakes**
- Improves **design thinking** and **coding practices**
- Enhances **code maintainability**

## 3. Cleanroom Technique

Developed by **IBM;** prevents defects instead of finding them later. Inspired by semiconductor "clean rooms."

### 🚫 Key Feature:

- **No code execution-based testing** allowed by the programmer.
- Only **formal verification**, **walkthroughs**, and **inspections** are used to remove bugs.

### ✅ Benefits:

- Highly **reliable and maintainable code**
- Strong emphasis on **defect prevention**

### ⚠️ Drawbacks:

- **Time-consuming**
- May miss **runtime** or **environment-specific** bugs that only show during real execution

# 7.3 SOFTWARE DOCUMENTATION

## 🎯 Purpose

Software documentation refers to all supporting documents created during the development lifecycle, alongside source code and executables.

***Documentation = all supporting texts accompanying software.***

## 📌 Importance of Documentation

- Helps **understand and maintain** code.
- Assists **users** in operating the software effectively.
- Mitigates the **manpower turnover** issue by ensuring knowledge transfer.
- Provides **project tracking evidence** to managers.

## Types of Documentation

## 1. internal Documentation

Located *within* the source code. Aimed at making the code more understandable to developers.

🛠️ **Common Forms:**

- **Comments** in code
- **Meaningful variable names** ✅ *Most useful* for understanding
- **Function/module headers**
- **Code indentation**
- **Modular structure**
- Use of:
  - **Enumerated types**
  - **Constant identifiers**
  - **User-defined data types**

⚠️ **Commenting Tip:**

Avoid trivial comments like:

```
a = 10;  /* a made 10 */  // ❌ Not helpful
```

Prefer comments that explain **non-obvious logic** or design decisions.

## 2. External Documentation

Created *outside* of code and shared with stakeholders.

### 📄 Examples:

- SRS (Software Requirements Specification)
- Design document
- Test document
- User manual
- Installation guide

### 🧭 Key Qualities:

- Must be **up-to-date** with code changes (to ensure **consistency**).
- Should be **understandable** by its target audience.

## 📏 Gunning's Fog Index

### 🧠 What is it?

A **readability metric** that tells how many years of education someone needs to understand a document.

❖ For example, a fog index of **12** means a person with **12th-grade education** can understand it comfortably.

### 🧮 Formula:

$$\text{fog}(D) = 0.4 \times \left( \frac{\text{Words}}{\text{Sentences}} \right) + \text{Per cent of words having 3 or more syllables}$$

➢ **Complex words**: Words with **3 or more syllables**

➢ **Syllables** are individual sounds, e.g., "understand" = *un-der-stand* (3 syllables)

## ✅ Example:

**Sentence**:

“The Gunning’s fog index is based on the premise that use of short sentences and simple words makes a document easy to understand.”

- Total words: **23**
- Complex words (≥3 syllables): **4**

### 🎯 Application:

If you’re writing a manual for users with **class 8 education**, ensure **fog index ≤ 8**.

**Count Complex Words**

A **complex word** means a word with **three or more syllables**.

In this sentence, the complex words (≥3 syllables) are:

*1.Gunning’s* – (Gun-ning’s → 2 syllables) ❌ Not complex

*2.premise* – (pre-mise → 2 syllables) ❌

*3.sentences* – (sen-ten-ces → 3 syllables) ✅

*4.simple* – (sim-ple → 2 syllables) ❌

*5.document* – (doc-u-ment → 3 syllables) ✅

*6.understand* – (un-der-stand → 3 syllables) ✅

*7.index* – (in-dex → 2 syllables) ❌

*8.based* – (1 syllable) ❌

So, there are **4 complex words** total.

Fog index = 0.4×(Average Sentence Length+%of Complex Words)
Since we have only one sentence:
**Average sentence length** = 23 words.
**% of complex words** = (4 ÷ 23) × 100 = 17.39%.
Now calculate:

$$\text{Fog } \textit{Index} = 0.4 \times (23 + 17.39)$$
$$= 0.4 \times 40.39 = 16.156$$

- ✓ A fog index of **16** means the document would be easily understood by someone with **16 years of formal education** (roughly a **college graduate**).
- ✓ **Simpler writing** (shorter sentences, fewer complex words) would lower the index and make it easier for a wider audience.

# 7.4 TESTING

## 🎯 Goal of Testing

- To identify and eliminate **defects** in software.
- However, **exhaustive testing is impossible** due to large input domains (e.g., floating-point numbers), but **careful testing** can still expose most bugs.

## 🔑 7.4.1 Basic Concepts & Terminologies

## 🧪 Testing Process

Involves:

1. Executing the program with **test inputs**.
2. Comparing **actual vs expected output**.
3. Noting down **failures and conditions** for **debugging**.

A highly simplified view of program testing is schematically shown in Figure 7.1.

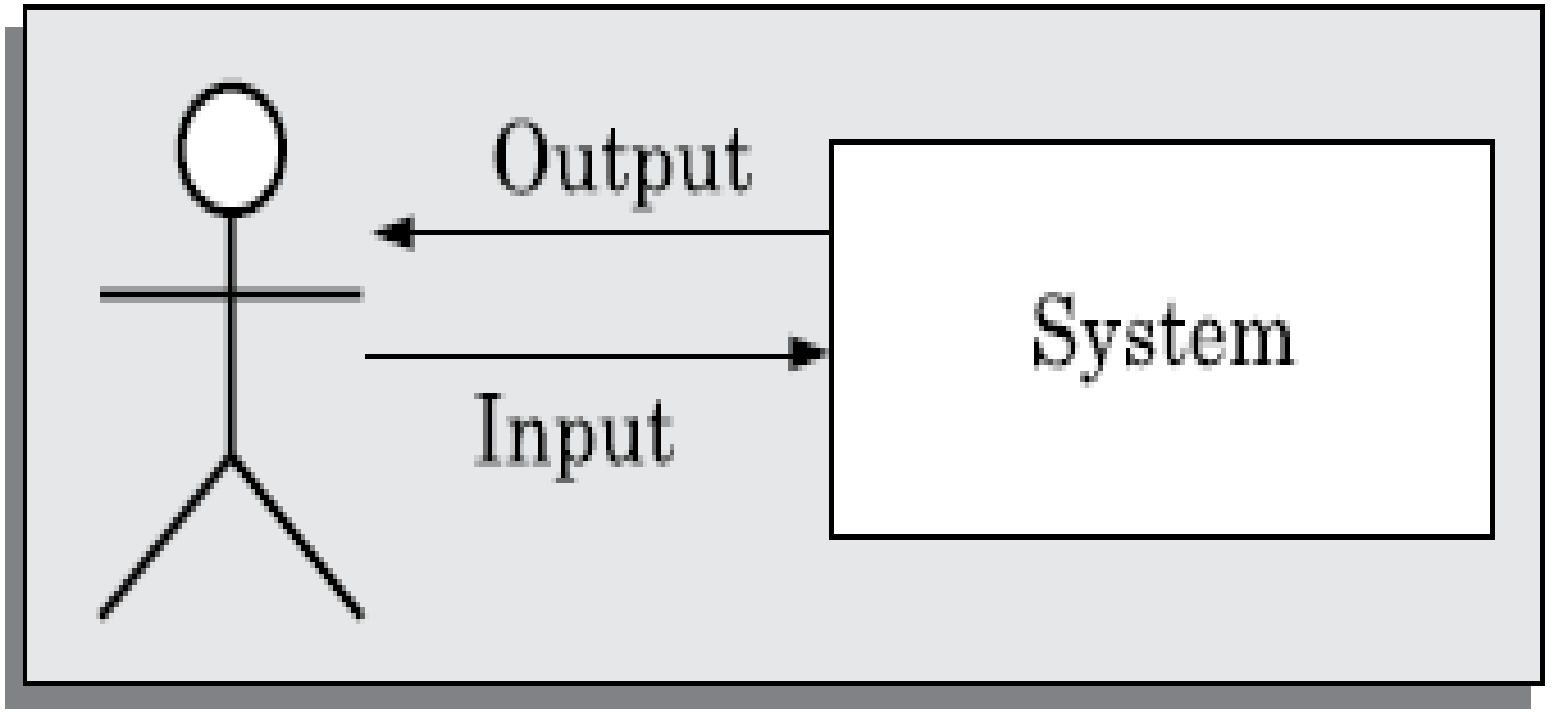



**FIGURE 7.1** A simplified view of program testing.

## 📘 Important

| Terms Term | Definition |
|---|---|
| Mistake | Human error during development (e.g., not initializing a variable). |
| Error / Fault / Bug / Defect | The result of a mistake. All considered synonymous. |
| Failure | Incorrect behavior during execution due to bugs. Example: Program crash or incorrect result. |
| Test Case | A triplet [Input (I), State (S), Expected Result (R)]. |
| Test Scenario | High-level idea of what to test; doesn't include specific input/output. |
| Test Script | Programmatic version of a test case for automation. |
| Positive Test Case | Tests correct functionality (e.g., correct login). |
| Negative Test Case | Tests undesired functionality (e.g., invalid login). |
| Test Suite | Collection of test cases. |
| Testability | How easily a system can be tested. |
| Failure Mode | Observable form of failure (e.g., crash, incorrect output). |
| Equivalent Faults | Different bugs that lead to the same failure (e.g., division by zero and illegal memory access → crash). |

## 🔄 Verification vs Validation

| Aspect | Verification | Validation |
|---|---|---|
| Purpose | Are we building the product right? | Are we building the right product? |
| Focus | Checks if one phase output conforms to the previous. | Checks if final software meets requirements. |
| Methods | Review, simulation, formal verification, unit/integration testing. | Primarily system testing. |
| Execution | Doesn’t require running code. | Requires software execution. |
| Example | Design vs SRS match. | Code vs user requirement match. |
| Goal | Early bug detection (phase containment). | Ensure software meets user needs. |

📝 ***Both together form the “V & V” (Verification and Validation) activities.***

A typical testing process in terms of the activities that are carried out has been shown schematically in Figure 7.2.

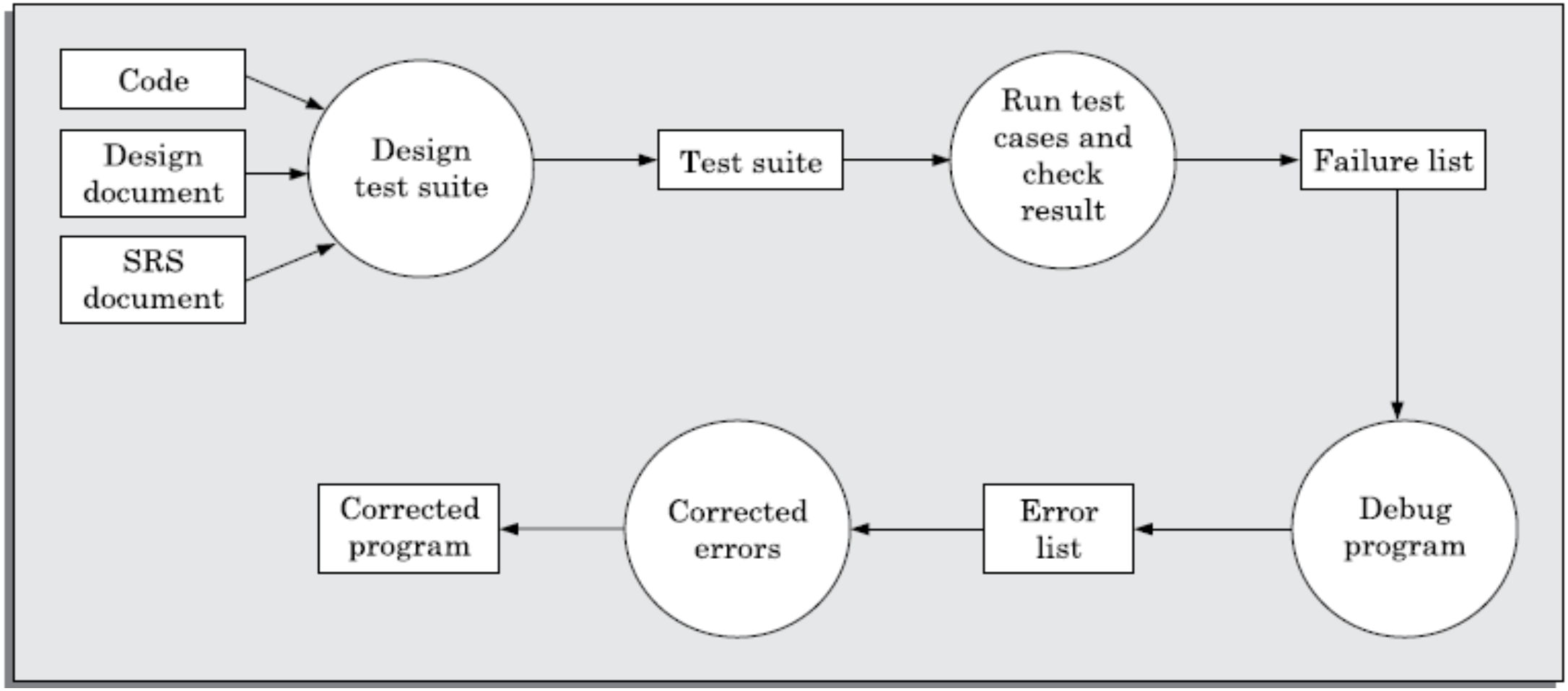



**FIGURE 7.2** Testing process.

## 🔁 7.4.2 Testing Activities

1. **Test Suite Design** – Create test cases using strategies.
2. **Run Test Cases** – Execute them and compare results.
3. **Locate Error (Debugging)** – Trace failures back to bugs.
4. **Correct Error** – Modify code and re-test.

🔍 *Debugging is often the most time-consuming activity.*

## ❓ 7.4.3 Why Design Test Cases?

- **Random inputs ≠ effective testing**.
- Carefully designed test cases find **unique errors**.
- Example: A wrong else block (max = x instead of max = y) may go undetected if only tested with inputs where x > y.

🧠 **Goal**: A **minimal**, well-designed test suite that finds **as many errors as possible**.

## 🎯 7.4.4 Testing in the Large vs Small

| Type | Definition |
|---|---|
| Unit Testing | Testing individual modules (functions/classes). |
| Integration Testing | Testing combined modules and their interactions. |
| System Testing | Testing the complete integrated system. |

- *Unit testing = Testing in the Small*
- *Integration + System testing = Testing in the Large*

## 📌 Why not test the system as a whole?

- Other modules may not be ready.
- Easier debugging at unit level.
- Helps localize errors faster.

## 7.4.5 Tests as Bug Filters

### 📉 Diminishing Returns

Once a test suite covers most defects, adding similar test cases won't help much. Each new **test strategy** filters more bugs, but **returns diminish**.

### 🐛 Bug Filter Analogy (Beizer, 1990):

Like using pesticides — first spray kills many bugs, survivors become resistant. You need **different sprays (test strategies)** to catch remaining bugs.

### 📐 Example Calculation – Bug Reduction

**Suppose: 10 test strategies, each removes 30% of current bugs.**
**Initial bugs = 1000**
**Remaining after each step = 70% of previous bugs**

$$\text{Bugs after 10 rounds} = 1000 \times (0.7)\,10 = 1000 \times 0.028 = 28 \text{ bugs left}$$

# Test Levels

- **Unit testing**
  - Test each module (unit, or component) independently
  - **Mostly done by developers of the modules**
- **Integration testing**
  - Modules are integrated in steps according to an integration plan
  - The partially integrated system is tested at each integration step
- **system testing**
  - Test the system as a whole
  - **Often done by separate testing or QA team**

# Overview of Activities During System and Integration Testing

- Test Suite Design
- Run test cases
- Check results to detect failures.
- Prepare failure list

**Tester**

- Debug to locate errors
- Correct errors.

**Developer**

# 7.5 UNIT TESTING

## Definition

Unit testing is the process of **testing individual modules or units** of software (functions, procedures, classes) **in isolation** to verify that they work correctly.

- Done **after coding** is complete and syntax errors are removed.
- Usually performed by the **developer (coder)** of that module.
- Conducted **during the coding phase**.

## Why Unit Testing Needs a Test Environment

To test a single module, it must be placed in a **suitable environment** that simulates its interactions with:

1. Other modules it **calls**
2. Global data it **uses**
3. A function that **calls it**

Since the complete system may not yet exist (other modules may not be ready), we use two artificial components:

## 🧩 Stubs and Drivers

### 🧰 Driver

- Acts as the **caller** of the module under test.
- Simulates the **higher-level module**.
- Provides:
  - **Input values**
  - Any **global or shared data**
  - Code to **invoke** the unit's functions

### 🔧 Stub

- Simulates the **lower-level modules** called by the unit under test.
- Is a **dummy procedure** with:
  - The **same interface** (same function signature)
  - **Simplified logic**, often returns fixed or table-based values

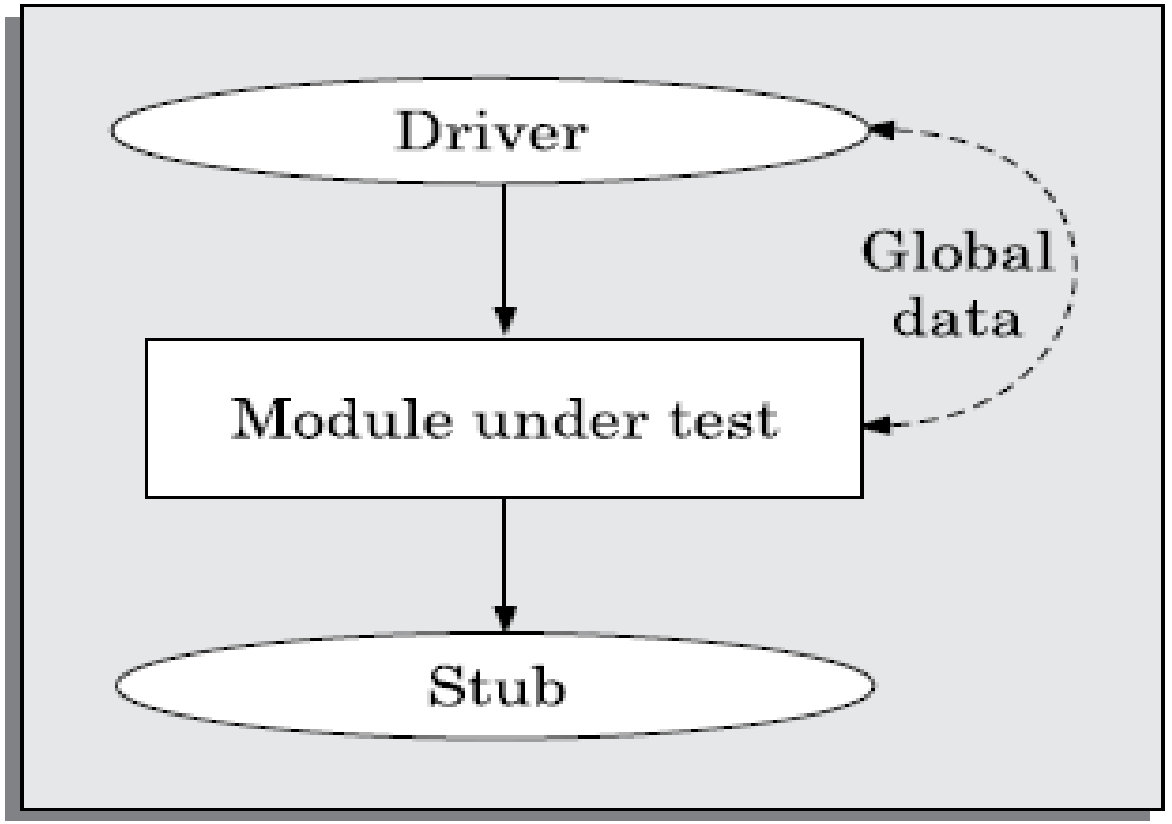



**FIGURE 7.3** Unit testing with the help of driver and stub modules.

Think of:

- **Driver** = Test Harness → Calls the module
- **Stub** = Fake Callees → Simulate dependencies

## ✅ Benefits of Unit Testing

- Helps detect **early-stage bugs** in individual functions.
- Makes **debugging easier**, as problems are localized.
- Encourages **modular design**.

## Limitation

Even well-designed unit test suites **cannot guarantee** that a module is completely error-free — only that it behaves as expected for the tested inputs.

## 📝 Example Scenario

If you're testing a calculateBill() function:

- The **Driver** might pass sample user and product data.
- The **Stub** for fetchDiscount() might return 10% always instead of computing from a database.

# 7.6 BLACK-BOX TESTING

## 🔍 What is Black-Box Testing?

- Test cases are **designed based only on input/output** specifications — **no knowledge of internal code or logic is required**.
- Also known as **functional testing**.

## 🔧 Two Main Techniques:

### 1. Equivalence Class Partitioning (ECP)

- **Idea**: Divide the input domain into **equivalence classes** such that:
  - All values within a class are **treated the same** by the program.
  - **Testing one representative value** from each class is **enough**.

### 🧠 How to Identify Equivalence Classes:

- If input is from a **range**, define:
  - **1 valid class** (within range)
  - **2 invalid classes** (below and above range)

• 📌 Example: For input range [1,10]

➤ Valid: [1–10]

➤ Invalid: (−∞ to 0), (11 to ∞)

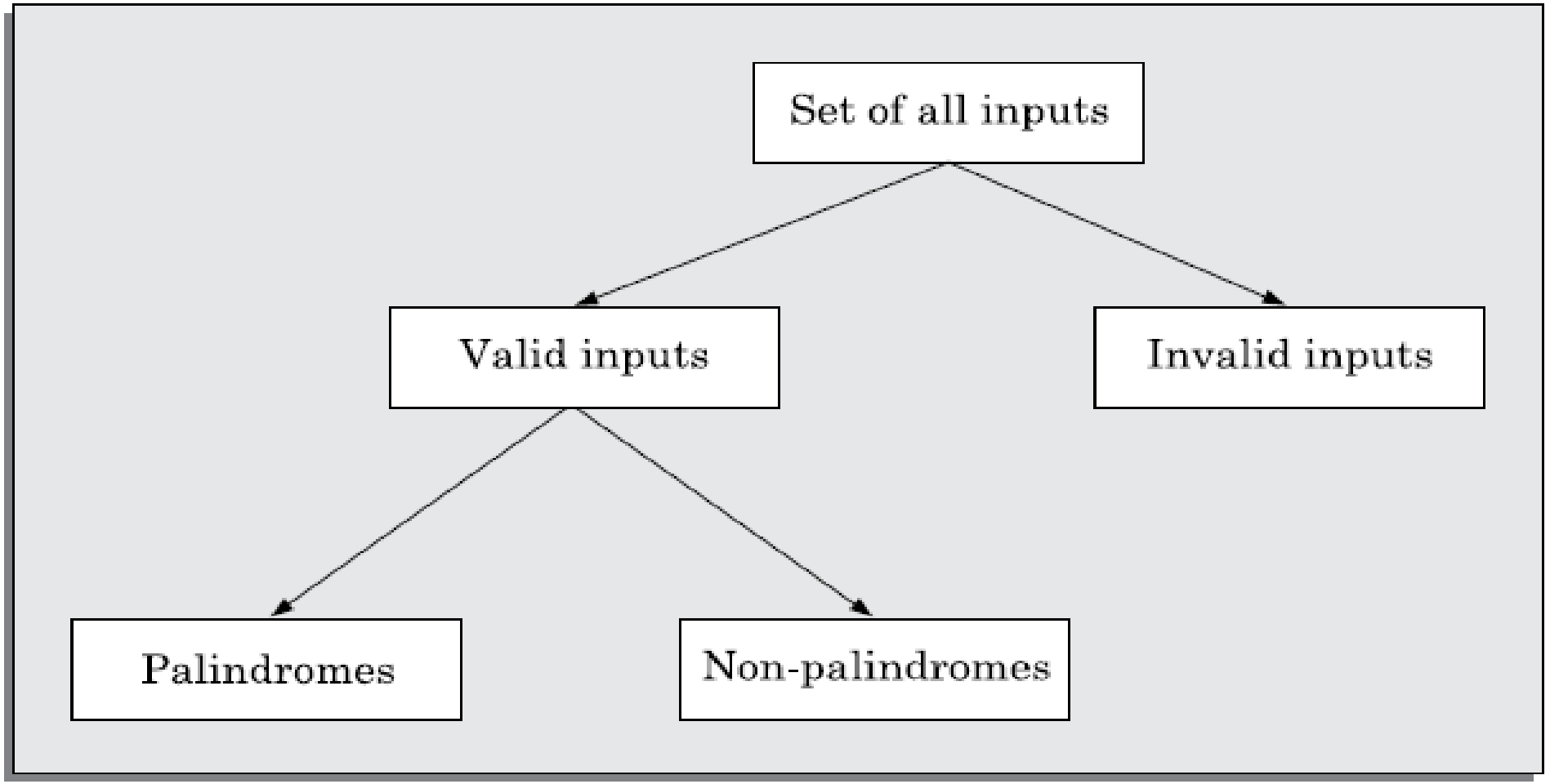



**FIGURE 7.4** Equivalence classes for Problem 7.3.

If input must come from a **specific set of valid values**, say {A, B, C}, then:
**Valid class:** {A, B, C}
**Invalid class:** All other inputs (anything not in {A, B, C})
This means we pick **one test value** from each class — one valid, one invalid — to represent the whole set.

✔️ **Examples:**
🧮 **Problem 7.1: Square root function for integers 0–5000**
**Specification:** Function accepts integers between 0 and 5000 (inclusive).
**Equivalence classes:**
Invalid: (-∞, -1] → negative numbers (can't take sqrt of negatives)
Valid: [0, 5000] → acceptable range
Invalid: [5001, ∞) → too large (beyond allowed range)
**Representative test cases:**
{ -5, 500, 6000 }
-5 → invalid (below range)
500 → valid
6000 → invalid (above range)

# 📏 Problem 7.2: Lines intersection

We are testing a **program that finds the intersection of two lines**.
When we input two lines, the program should tell us **what kind of intersection** they have.
There are **three possible outcomes**:
1.The lines are **parallel** → they never meet.
2.The lines are **intersecting** → they meet at **exactly one point**.
3.The lines are **coincident** → they are the **same line**, so they meet at **infinite points**.

## 🧠 Step 1: Identify the Classes

Each type of intersection represents a **different equivalence class** — because the program's behavior (output) changes depending on which case it is.
So the equivalence classes are:

| Class Name | Meaning | Example Behavior |
|---|---|---|
| **Parallel** | Lines never meet | Output: "No intersection" |
| **Intersecting** | Lines cross at one point | Output: "Lines meet at (x, y)" |
| **Coincident** | Lines overlap completely | Output: "Infinite intersections" |

**Step 2: Choose Representative Test Cases**
We don’t need to test *every possible pair of lines* — just **one example per class**, because all lines in the same class behave similarly.

| Class | Test Case Example | Explanation |
|---|---|---|
| Parallel | `{(2,2)(2,5)}` | Two vertical lines that never meet. |
| Intersecting | `{(5,5)(7,7)}` | Two lines crossing each other. |
| Coincident | `{(10,10)(10,10)}` | Both lines are exactly the same. |

🧾 **Case 1: Parallel lines — `{(2,2)(2,5)}`**
These are **two vertical lines** at **x = 2** and **x = 5** (for example).
They go up and down but never meet.
🧱 Class: **Parallel (no intersection)**
🧪 Expected Output: "No intersection point."

🗒️ **Case 2: Intersecting lines — `{(5,5)(7,7)}`**

These are **two lines with different slopes**, so they cross each other at exactly one point.

Think of one line going diagonally up-right and another going diagonally down-right — they cross once.

🧱 Class: **Intersecting (one intersection point)**

🧪 Expected Output: "Intersection at (x, y)."

🗒️ **Case 3: Coincident lines — `{(10,10)(10,10)}`**

Here, the two lines are **exactly the same** — all their points overlap.

That means they meet at **infinite points**.

🧱 Class: **Coincident (infinite intersections)**

🧪 Expected Output: "Lines are the same."

You are dividing the behavior of "lines intersection" into **three logical categories (equivalence classes)** and picking **one test case from each**.
That's how **Equivalence Class Partitioning** works — each test represents a group of similar possible inputs.

## 🔁 Problem 7.3: Palindrome checker

**Specification:** Input string must have **fewer than 5 characters**.

**Equivalence classes:**

- Valid Palindrome (e.g., "aba")
- Valid Non-palindrome (e.g., "abc")
- Invalid Input (≥5 chars, e.g., "abcdef")

**Test cases:** { "aba", "abc", "abcdef" }

- "aba" → palindrome → valid
- "abc" → non-palindrome → valid
- "abcdef" → invalid input (too long)

### 🎯 Key Idea

Each equivalence class represents a **distinct type of input behavior**.
By testing one value from each class, we efficiently ensure **broad coverage** of all functional possibilities — this is the essence of **Equivalence Class Partitioning** in **black-box testing**.

## 2. Boundary Value Analysis (BVA)

- **Idea**: Programmers often make mistakes at the **boundaries** of input ranges (e.g., using < instead of <=).
- So, test cases should include:
  - **Lower bound**, **upper bound**, and values **just outside** these bounds.

## ✔ Examples:

- **Problem 10.11**: Square root for range [0–5000]
  - ➤ Test cases: {−1, 0, 5000, 5001}

- **Problem 10.12**: String palindrome (< 5 characters)
  - ➤ Boundary between valid and invalid input length
  - ➤ Test cases: {abcdef (6 chars), abcde (5 chars)}

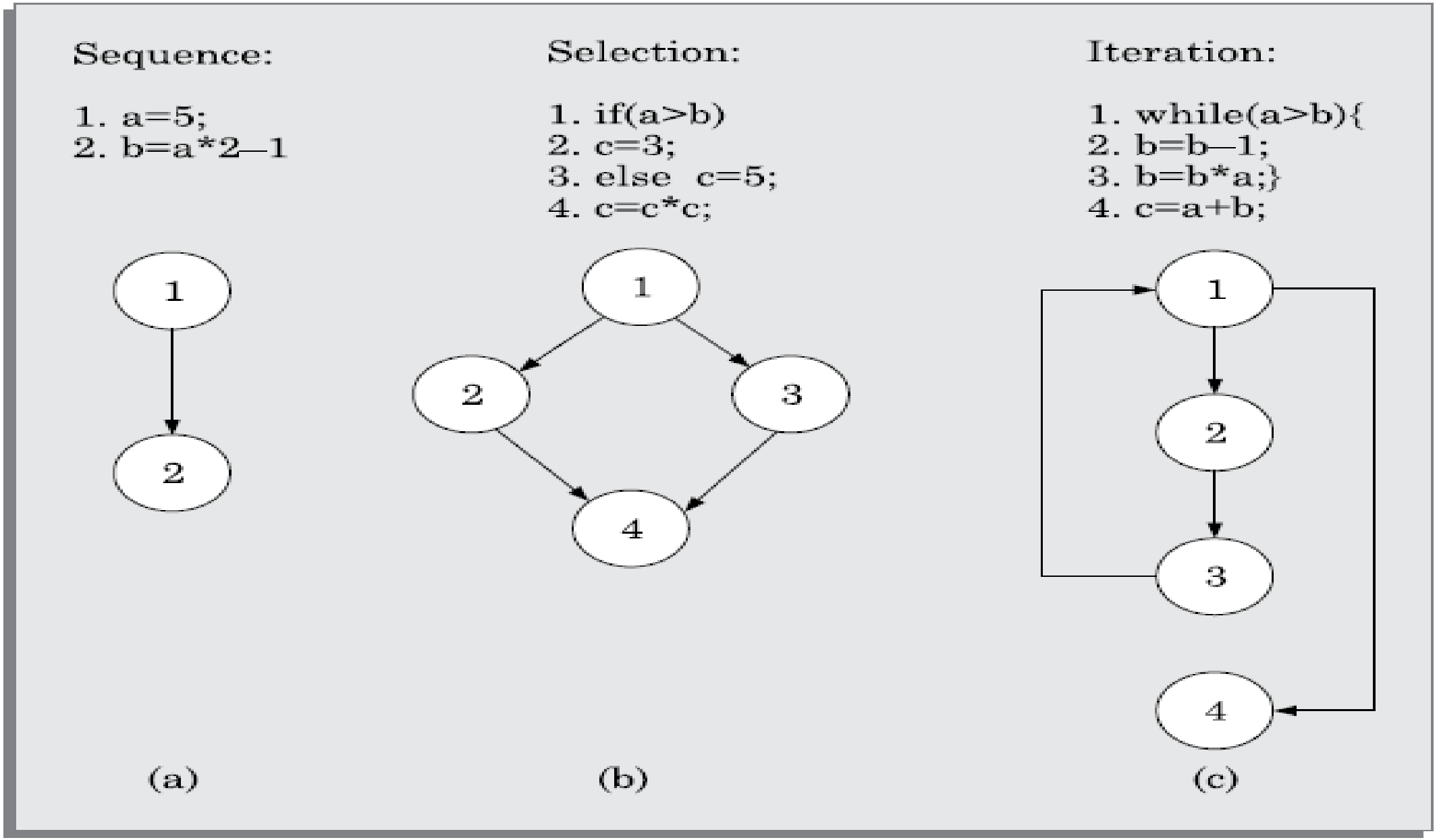



**FIGURE 7.5** CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

## 📝 Summary of Black-Box Testing Process:

1. **Examine** input and output values of the software.
2. **Identify** equivalence classes (valid & invalid).
3. **Select** one representative test input from each equivalence class.
4. **Design** boundary value test cases by:
    1. Including edge values
    2. Including just-beyond-the-edge values

- **Equivalence Class Testing** → Maximizes coverage with minimal test cases.
- **Boundary Testing** → Targets common edge-case errors.
- Black-box testing is **simple, intuitive**, but **requires practice** to identify all equivalence classes accurately.
- Use both ECP and BVA together for **effective black-box testing**.

# 7.7 WHITE-BOX TESTING

White-box testing is a **unit testing method** based on **internal code structure**. It uses the source code to design test cases. There are **two types** of strategies:

## 📌 7.7.1 Basic Concepts

- **Coverage-Based Testing**: Ensures specific parts of code (e.g., statements, branches) are executed.
- **Fault-Based Testing**: Designed to uncover specific faults (e.g., using mutation testing).

A test suite is **adequate** if it satisfies the chosen **testing criterion**.

**Comparison**:

- One strategy is **stronger** if it covers all that another does and **more**.
- Two strategies are **complementary** if they cover different program elements.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called *complementary testing strategies*. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 7.6. Observe in Figure 7.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 7.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and *vice versa*.

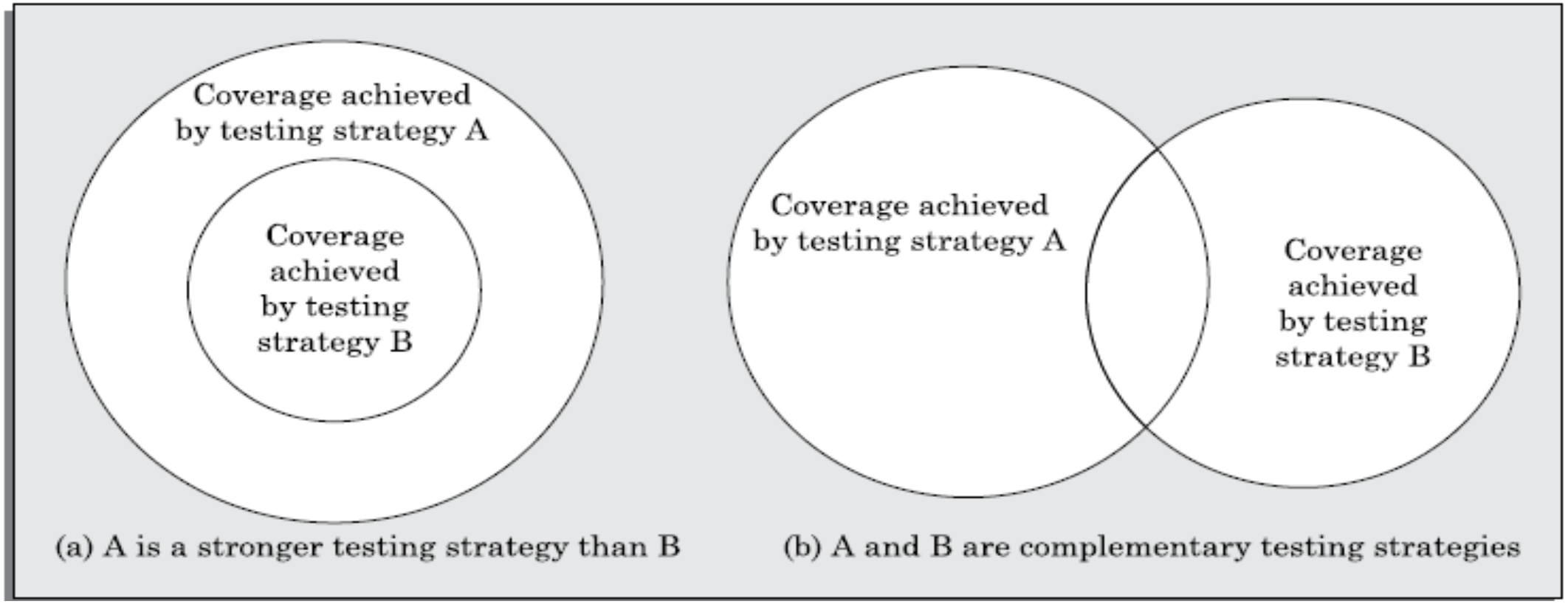



**FIGURE 7.6** Illustration of stronger, weaker, and complementary testing strategies.

## 7.7.2 Statement Coverage

Statement coverage is a white-box testing technique that ensures **every executable statement in the source code is executed at least once** during testing.

To verify that no part of the code remains untested.

**Formula:**

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

**Strength:**

✓ Ensures that all lines of code have been run at least once.

✓ Helps identify **dead code** or **unreachable statements**.

**Weakness:**

- It **does not ensure full logic validation** — some logical conditions may never be tested even if all statements are executed.
- Cannot detect missing paths or conditions that affect control flow.

# Example GCD

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

**Test Cases for Statement Coverage:**

`1.(3, 3)` → Executes the return statement directly (loop skipped).

`2.(4, 3)` → Executes `if (a > b)` branch.

`3.(3, 4)` → Executes the `else` branch.

✅ All statements are executed at least once.

**Limitation:**

These test cases cover all statements but **do not guarantee all logical conditions** (`a > b` and `a < b`) are tested in all possible orders or scenarios.

## 7.7.3 Branch Coverage (Decision Coverage)

Branch coverage, also known as **decision coverage**, ensures that **each possible branch (True and False outcome)** of every decision point (like if, while, or for) in the program is executed **at least once** during testing.

To verify that all decision outcomes in the code are tested.

**Formula:**

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

**Strength:**

Stronger than **Statement Coverage** because it tests **both True and False outcomes** of each decision.

Helps identify missing conditions or untested logical paths.

**Weakness:**

Does not ensure that **all combinations** of conditions in complex Boolean expressions are tested (that requires *Condition Coverage*).

May still miss certain logical errors.

# Example: Euclid's GCD Algorithm

```
int gcd(int a, int b) {
   while (a != b) {
      if (a > b)
         a = a - b;
      else
         b = b - a;
   }
   return a;
}
```

**Decision Points:**

`1.while (a != b)` → True / False

`2.if (a > b)` → True / False

| Test Case | Description | while (a!=b) | if (a>b) |
|---|---|---|---|
| (3,3) | Loop condition false immediately | False | — |
| (3,2) | Loop true, if true branch executes | True | True |
| (4,3) | Loop true, if true branch executes | True | True |
| (3,4) | Loop true, else branch executes | True | False |

✅ **All True and False branches** of both decisions are executed at least once.

Thus, **Branch Coverage = 100%**

## 7.7.4 Condition Coverage (BCC)

Condition coverage (also called **Basic Condition Coverage**) ensures that **each individual condition** within a decision (like if, while, or for) has been evaluated to **both True and False** at least once — **independently of the overall decision outcome**. To verify that **all atomic Boolean expressions** inside decisions are tested for both outcomes (True and False).

**Formula:**

$$\text{Condition Coverage} = \frac{\textit{Number of condition outcomes executed (T/F)}}{\textit{Total number of condition outcomes}} \times 100$$

**Strengths:**

Ensures every simple Boolean sub-condition (e.g., a > b, x == 0, etc.) is evaluated as both **True** and **False**.

Detects logical errors hidden within complex compound conditions.

**Weaknesses:**

Does **not ensure that all decision outcomes** (True/False) are covered — some combinations of conditions might never make the overall decision True or False.

Hence, **Condition Coverage ≠ Branch Coverage**.

**Example:**

```
if ((a > b) && (b > 0))
   printf("Valid\n");
else
   printf("Invalid\n");
```

**Basic Conditions:**

`1.a > b`
`2.b > 0`

Each of these two conditions must be **True** and **False** at least once, regardless of the overall decision result.

| Test Case | a | b | (a>b) | (b>0) | Overall Decision |
|---|---|---|---|---|---|
| 3, 2 | T | T | True | | |
| 3, -1 | T | F | False | | |
| 1, 2 | F | T | False | | |
| 1, -1 | F | F | False | | |

✅ Each basic condition (`a>b` and `b>0`) has been **True and False** at least once.
❌ However, not all combinations necessarily make the **overall decision True or False**, so **full branch coverage is not guaranteed**.

## 7.7.5 Condition and Decision Coverage

**Condition and Decision Coverage (CDC)** is a combination of both **Condition Coverage** and **Decision (Branch) Coverage**.

It ensures that:

**Each basic condition** in a decision takes on **both True and False** values at least once, **and**

**Each decision (overall outcome)** itself evaluates to **True and False** at least once.

To achieve better logic testing by covering both **individual conditions** and **overall decision outcomes**.

**Formula:**

$$\text{Condition } \textit{and Decision Coverage} = \frac{\textit{Number of conditions (T/F) and decisions (T/F) executed}}{\textit{Total number of conditions and decisions}} \times 100$$

**Strengths:**

Combines the benefits of **Condition Coverage** and **Branch Coverage**.

Detects more logical errors compared to either of them individually.

Provides higher confidence in the correctness of decision-making logic.

**Weaknesses:**

May still not test **all combinations** of conditions (for that, *Multiple Condition Coverage* or *Modified Condition/Decision Coverage – MC/DC* is needed).

Increases the number of required test cases compared to basic branch testing.

Example:

```
if ((a > b) && (b > 0))
   printf("Valid\n");
else
   printf("Invalid\n");
```

**Basic Conditions:**

1. `a > b`
2. `b > 0`

**Decision:** `(a > b) && (b > 0)`

| Test Case | a | b | (a > b) | (b > 0) | Decision Result |
|---|---|---|---|---|---|
| 3, 2 | T | T | True | | |
| 3, -1 | T | F | False | | |
| 1, 2 | F | T | False | | |
| 1, -1 | F | F | False | | |

✅ Each **basic condition** has been **True** and **False** at least once.
✅ The **overall decision** has evaluated to **True** and **False**.
Thus, **Condition and Decision Coverage = 100%**.

| Coverage Type | Ensures | Notes |
| --- | --- | --- |
| Statement Coverage | Every statement executed | Weakest |
| Branch Coverage | Every decision True/False | Stronger |
| Condition Coverage | Each condition True/False | Does not ensure decision outcomes |
| **Condition & Decision Coverage** | Both individual conditions and overall decisions True/False | Stronger, more thorough |

## 7.7.6 Multiple Condition Coverage (MCC)

Multiple Condition Coverage ensures that **all possible combinations** of the **atomic conditions** in a decision are evaluated **at least once**.
It is the most thorough form of logical coverage, testing every possible **True/False combination** of the conditions involved.

**Key Points:**

Each decision is made up of one or more **atomic conditions** (simple Boolean expressions).
MCC requires testing $\mathbf{2^n}$ combinations, where **n** = number of atomic conditions.
It ensures that **every possible logical path** through the decision is verified.

**Formula:**

If a decision has *n* conditions,

$$\text{Number } \textit{of test cases} = 2^n$$

**Example:**

Decision:

$$(temperature > 150) \;||\; (temperature > 50)$$

Here,

- Condition 1: $C1 = (temperature > 150)$
- Condition 2: $C2 = (temperature > 50)$

→n = 2 → $2^2$ = **4 test cases** required.

| Test Case | C1 (temp >150) | C2 (temp >50) | Decision Output |
|---|---|---|---|
| 1 | F | F | F |
| 2 | F | T | T |
| 3 | T | F | T |
| 4 | T | T | T |

## 7.7.7 Modified Condition/Decision Coverage (MC/DC)

Modified Condition/Decision Coverage (MC/DC) ensures that **each atomic condition within a decision** is shown to **independently affect the outcome** of that decision.
It provides **almost the same fault-detection power as Multiple Condition Coverage (MCC)** but requires **significantly fewer test cases** (linear in *n* instead of exponential).

**Key Points:**

Each **decision** must evaluate to both **True** and **False**.
Each **atomic condition** must take on both **True** and **False** values.
Each **condition** must be shown to **independently influence** the decision's result (i.e., changing only that condition flips the decision outcome).

**Why It's Important:**

Strikes a **balance between thoroughness and practicality**.
Widely used in **safety-critical systems** such as:

- **Avionics (FAA DO-178C)**
- **Automotive (ISO 26262)**
- **Medical devices and aerospace systems**

## Subsumption hierarchy

We have already pointed out that MCC subsumed the other condition-based test coverage metrics. In fact, it can be proved that we shall have the subsumption hierarchy shown in Figure 7.7 for the various coverage criteria discussed. We leave the proof to be worked out by the reader. Observe that MCC is the strongest, and statement and condition coverage are the weakest. However, statement and condition coverage are not strictly comparable and are complementary.

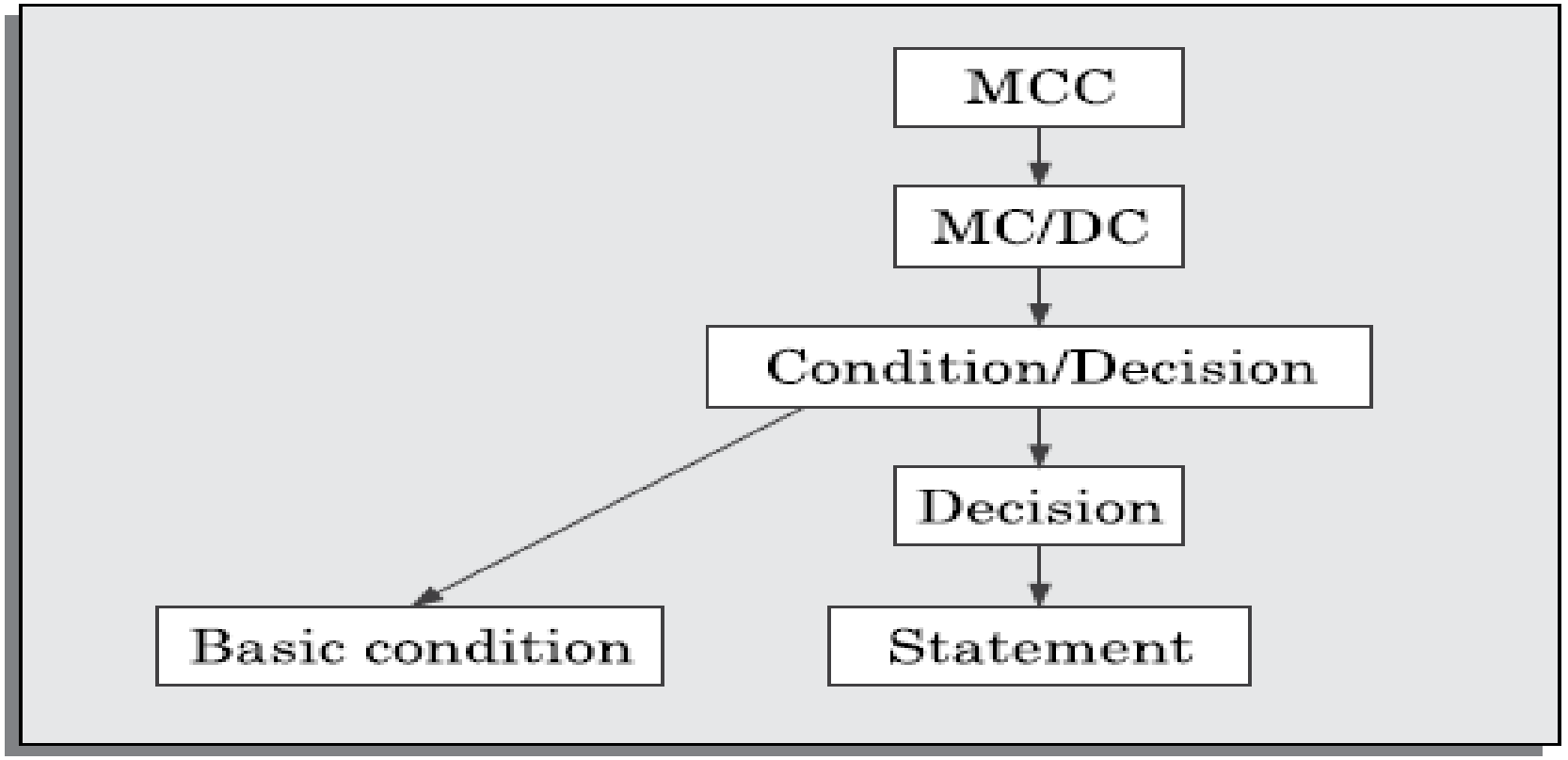



**FIGURE 7.7** Subsumption hierarchy of various coverage criteria.

The figure illustrates how various **condition-based test coverage criteria** relate to one another in terms of **strength (or thoroughness)**.

- **Multiple Condition Coverage (MCC)** is the **strongest** criterion — it subsumes all the others because it requires testing **all combinations** of conditions.

- **Modified Condition/Decision Coverage (MC/DC)** is slightly weaker than MCC but still powerful, ensuring each atomic condition independently affects the decision outcome.

- **Condition/Decision Coverage** combines both decision and condition testing requirements.

- **Decision Coverage** ensures every decision's outcome (True/False) is executed at least once.

- **Basic Condition Coverage** verifies that each individual condition is evaluated both True and False, but not necessarily in combination.

- **Statement Coverage** is the **weakest**, ensuring that every line or statement of code is executed at least once.

## 📈 7.7.8 Path Coverage

- Covers all **linearly independent paths** in the control flow graph (CFG).
- Relies on **McCabe's Cyclomatic Complexity (V(G))**:
  - **V(G) = E – N + 2**
  - E: Edges, N: Nodes in CFG.
  - Also: V(G) = Number of decision points + 1

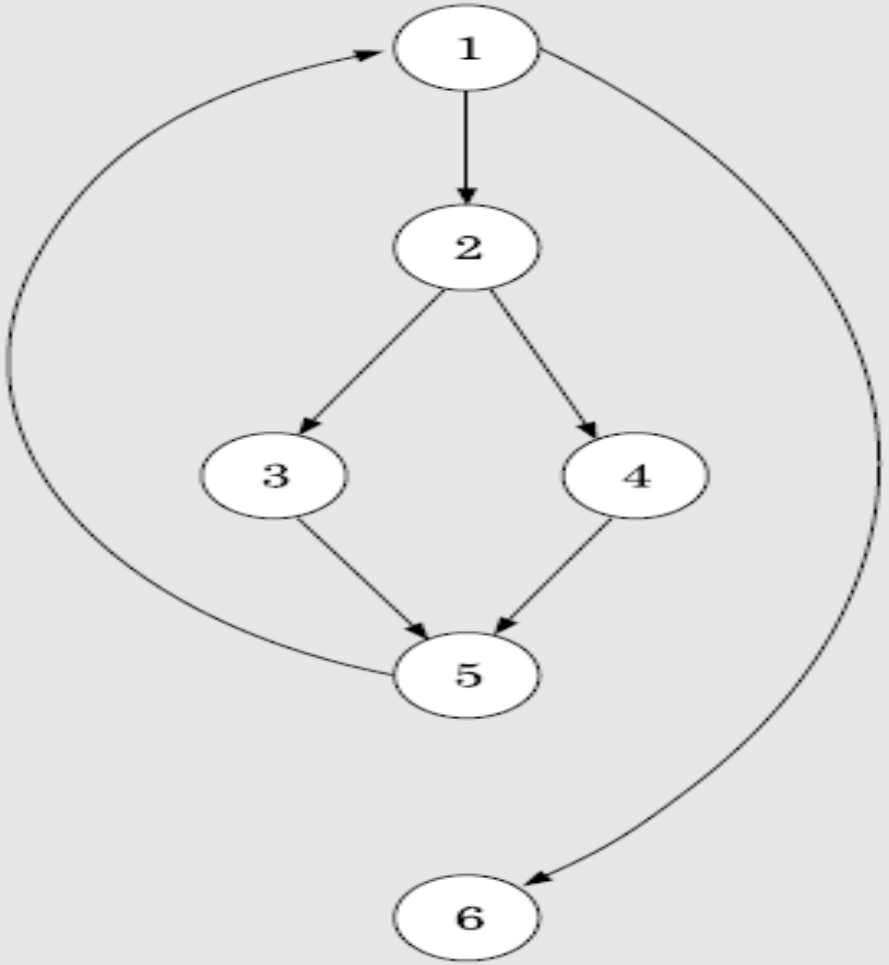
**Steps for Path Testing**:

1. Draw CFG.
2. Compute V(G).
3. Random test cases + dynamic analysis until ~90% path coverage is achieved.

Using these basic ideas, the CFG of the program given in Figure 10.8(a) can be drawn as shown in Figure 10.8(b).

```
int compute_gcd(int x, int y) {
   1  while(x!=y) {
   2        if(x>y) then
   3           x=x-y;
   4        else y=y-x;
   5     }
   6     return x;
     }
```

(a) An example program




(b) Control flow graph

**FIGURE 7.8** Control flow diagram of an example program.

## 7.7.9 McCabe's Cyclomatic Complexity Metric

McCabe's Cyclomatic Complexity is a **graph-based software metric** that provides:

- An **upper bound on the number of linearly independent paths** through a program.
- A **quantitative measure of code complexity**, helping estimate **testing effort**, **code maintainability**, and **reliability**.

## Three Methods to Compute Cyclomatic Complexity

Let V(G) be the cyclomatic complexity of the control flow graph (CFG) G.

**Method 1: Using Nodes and Edges**

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

- E: Number of edges in the CFG
- N: Number of nodes in the CFG

📌 *Example:*

If E = 7, N = 6 → V(G) = 7 - 6 + 2 = 3

## Method 2: Using Bounded Areas

**V(G) = Number of bounded areas + 1**

➢ Count the distinct **non-overlapping bounded regions** (closed loops) in a **planar CFG**.

📌 *Example:*

2 bounded areas → V(G) = 2 + 1 = 3

🔴 **Note:** This method does **not work** for **non-planar graphs** (e.g., those with goto statements).

## Method 3: Using Decision/Loop Statements

V(G)=Number of decision/loop statements+1

•Count all if, while, for, switch, etc.

📌 *Example:*

2 decisions → V(G) = 2 + 1 = 3

**Count the following statements:**

✅ `if`
✅ `else if`
✅ `while`
✅ `for`
✅ `do-while`
✅ `switch`
✅ Conditional operator `? :` (in some cases)

```
if (x > 0)
    y++;
else
    y--;
while (y < 10)
    y++;
```

Here:

- `if` → 1 decision
- `while` → 1 loop

Total decisions/loops = 2

$$V(G) = 2 + 1 = 3$$

## ✅ Interpretation

- A higher V(G) indicates **more complex code**, requiring **more test cases**.
- Typically, V(G) ≤ 10 is acceptable. Beyond that, functions are considered **too complex**.

## 🧪 Using McCabe's Metric for Path Testing

### 🚶 Steps to Perform Path Coverage Testing

1. **Draw** the control flow graph (CFG) of the program.
2. **Compute** cyclomatic complexity V(G) using one of the above methods.
3. **Determine** the minimum number of test cases needed = V(G).
4. **Generate** test cases (often random inputs).
5. **Execute** and monitor path coverage with tools (like dynamic program analyzers).
6. **Repeat** until ≥90% path coverage is achieved.

🟡 *Note:* Full 100% path coverage is **not practical** due to:

- **Infeasible paths** (e.g., contradictory conditions)
- **Exponential increase** in path count with code complexity

## 📌 Uses of McCabe's Cyclomatic Complexity

| Use | Purpose |
|---|---|
| Structural Complexity Estimation | Indicates code difficulty and maintainability |
| Testing Effort Estimation | Cyclomatic complexity ≈ Minimum test cases required |
| Reliability Estimation | Higher complexity ⇒ Higher chance of undetected errors |

## ✅ Best Practice:

- Keep function complexity under **10** for maintainability
- Limit complexity to **≤7** to reduce **testing effort**

- **Cyclomatic Complexity** measures the **logical complexity** of code.
- It helps in estimating **testing efforts**, understanding **code structure**, and ensuring **software reliability**.
- It is widely used in **path-based testing** to determine **minimum required test cases**.

## 7.7.10 Data Flow-Based Testing

💡 **Purpose:** To ensure that **data (variables)** in a program are **defined, used, and killed (redefined)** properly — and that no variable is used **before initialization** or **after its value becomes invalid**. Focuses on **how variables are defined and used** in the code, aiming to test **data usage across paths** in a program.

✅ **Key Concepts:**

- **DEF(S):** Variables **defined** at statement S

  → e.g., for a = b + c;, DEF(S) = {a}

- **USES(S):** Variables **used** at statement S

  → e.g., for a = b + c;, USES(S) = {b, c}

❖ A definition is **live** at a statement S1 if there's a **path from S to S1** with **no redefinition** of that variable.

## 🔍 Test Coverage Criteria in Data Flow Testing:

| Criterion | Meaning |
|---|---|
| All-Definitions Criterion | At least one use of each variable definition must be tested. |
| All-Uses Criterion | All uses of each variable definition must be tested. |
| All DU-Paths Criterion | Test every path between a variable's definition and its use, excluding complex cycles. |
| DU Chain Coverage | Each Definition-Use (DU) chain must be tested at least once. |

## 🔗 DU Pair and DU Chain (Definition-Use Chain (DU Chain)):

- **DU Pair:** [X, S, S1] → Variable X is defined at S, used at S1, and is live in between.
- **DU Chain:** A sequence of DU pairs.

🟡 ***Used to ensure the data is handled correctly from assignment to use across the program's execution.***

Example:

1. a = 10;  // Definition of a
2. b = a + 5;  // Use of a (DU chain: $a_1 \rightarrow a_2$)
3. a = 20;  // Redefinition (kills previous definition)

## Live Definition Example:

S1: a = 5;  → defines a
S2: if (a > 0)  → uses a (a is live here)
S3: a = 7;  → redefines a (previous definition no longer live)

⚙️ **Purpose of Data Flow Testing**

- Detects **data anomalies** such as:
  - **Undefined variable use**
  - **Unreferenced definitions**
  - **Multiple redundant definitions**
- Ensures that each defined variable is used meaningfully.

## 7.7.11 Mutation Testing

### 💡 Purpose:

A **fault-based testing** method to evaluate the **effectiveness of a test suite** by introducing **small code errors** (mutants).

### ✅ Key Concepts:

- A **mutant** is a version of the program with a **small change**, like:
  - + → -
  - && → ||
  - Deleting a statement
  - Changing variable type or value
- If a test case detects a difference (i.e., fails on the mutant), the **mutant is killed**.

## 🧪 Process of Mutation Testing:

1. **Initial Testing**: Test the original program with existing test cases.
2. **Mutant Generation**: Apply **mutation operators** to create mutants.
3. **Testing Mutants**: Run original test suite on each mutant.
4. **Evaluate:**
    1. **Killed mutant**: Detected by test suite (good).
    2. **Alive mutant**: Not detected (test suite is weak → improve tests).
    3. **Equivalent mutant**: Functionally same as original (cannot be killed).

## ⚙️ Mutation Operators Examples:

- Replace + with -
- Replace > with <
- Change constant 5 to 0
- Delete a line or variable

## 📈 Advantages:

- Reveals weaknesses in test suites.
- Automatable using mutation tools (e.g., PIT, MutPy).

## ⚠️ Challenges:

- **High computational cost**
- **Many mutants** generated
- **Difficult** to detect **equivalent mutants**

# 7.8 DEBUGGING

## 💡 Definition:

Debugging is the process of **identifying, locating, and fixing errors (bugs)** in a program after a **failure** has been detected.

## 7.8.1 Debugging Approaches

- Different strategies can be used to trace the cause of an error. Each has its **pros and cons** depending on the complexity and nature of the program.

## 1. Brute Force Method

- **What it is**: Adding print statements to check values at various points.
- **Advanced tool**: Use a **symbolic debugger** to check variable values, set breakpoints, and step through code.
- **Form**: Single-stepping with expected vs actual results.
- **Drawback**: **Inefficient** and time-consuming, especially for large programs.

## 2. Backtracking

- **What it is**: Trace code **backward** from where failure occurred to find the bug.
- **Drawback**: Becomes **complex** and **impractical** with **loops, branches**, and long control paths.

## 3. Cause Elimination Method

- **What it is**: List **possible causes** for the error and eliminate them **one by one** through testing.
- **Technique**: Similar to **fault tree analysis**.
- **Strength**: Good for **systematic narrowing** of fault source.

## 4. Program Slicing

- **What it is**: Identify **only the relevant code lines** affecting the value of a variable.
- **Definition**: A **slice** is the set of statements that influence the value of a variable at a particular point.
- **Advantage**: **Reduces search space** compared to full backtracking.

## 7.8.2 Debugging Guidelines

Here are some key **best practices** for effective debugging:

| Guideline | Why it matters |
|---|---|
| ✅ Understand the program design completely | Partial understanding can mislead debugging efforts. |
| ⚠️ Avoid fixing only the symptoms | Might hide the real bug and cause deeper issues later. |
| 🔁 Be prepared to redesign if necessary | Sometimes, poor design is the root cause, not just faulty logic. |
| 🔄 Always do regression testing after fixing bugs | To ensure that new bugs aren't introduced while fixing old ones. |

## 7.9 PROGRAM ANALYSIS TOOLS

### ✅ Definition:

Program analysis tools are **automated software tools** that analyze a program's **source or executable code** to assess various attributes such as:

- Size
- Complexity
- Comment quality
- Coding standard adherence
- Testing adequacy

These tools help **improve code quality** and ensure **reliable software** development.

### 🔸 TYPES OF PROGRAM ANALYSIS TOOLS

Program analysis tools are broadly classified into:

| Type | Description |
|---|---|
| Static Analysis Tools | Analyze code without executing it |
| Dynamic Analysis Tools | Analyze program during execution using real inputs |

## 7.9.1 Static Analysis Tools

### 📌 What They Do:

Static analysis tools check the **source code** to:

- Calculate metrics: Cyclomatic complexity, Halstead's metrics, size
- Check **coding standard compliance**
- Detect **common programming errors**, e.g.:
  - Uninitialized variables
  - Unused variables
  - Mismatched function parameters

### ✅ Examples of Static Analysis Activities:

- Code walkthroughs
- Code inspections

(These are **manual static methods**, not automated tools)

## ⚠️ Limitations:

- Cannot evaluate **run-time behaviour**
- Cannot handle **dynamic memory** issues caused by:
  - Pointer arithmetic
  - Dynamic memory allocation (e.g., using malloc in C)

## 📈 Kiviat Chart (a.k.a. Radar Chart):

- A graphical tool used to **summarize analysis** of each function
- Axes show:
  - Cyclomatic complexity
  - Lines of code
  - % of comment lines
  - Halstead metrics, etc.

## 7.9.2 Dynamic Analysis Tools

### 📌 What They Do:

Dynamic analysis tools monitor the **program's behaviour during execution** to:

- Evaluate **test coverage**
- Record execution trace (values of variables, paths taken, etc.)

### ⚙️ How It Works:

- Uses **instrumentation**: Adds extra code to log behaviour during execution
- Collected data includes:
  - Path coverage
  - Statement coverage
  - Branch coverage

### ✅ Use Cases:

- Analyze and improve **test suite effectiveness**
- Identify **redundant** or **missing** test cases
- Ensure **sufficient coverage** has been achieved
- Provide **evidence** of proper testing (e.g., printed reports, graphs)

## 📊 Output Format:

- Coverage results are often shown using:
  - **Histograms**
  - **Pie charts**
  - **Reports per module**

| Feature | Static Analysis Tools | Dynamic Analysis Tools |
|---|---|---|
| Execution Required? | ❌ No | ✅ Yes |
| Detects Run-time Issues? | ❌ No | ✅ Yes |
| Useful for Early Detection | ✅ Yes | ❌ No (Requires executable/test cases) |
| Typical Output | Kiviat charts, metrics, error lists | Histograms, pie charts, coverage reports |
| Examples of Use | Coding standard checks, complexity stats | Path coverage, identifying weak test cases |

# Regression Testing

## 🔸 Definition:

**Regression Testing** is the process of re-running previously executed test cases **after a change** is made to the code (e.g., bug fix or enhancement) to ensure that:

- Existing functionalities **still work correctly**
- **No new bugs** are introduced as a side-effect

## 📍 Key Characteristics:

- **Spans multiple levels**: Regression testing is **not limited** to unit, integration, or system testing—it's an **orthogonal** (separate) activity that applies across all these levels.
- Must be **repeated** after:
  - Bug fixes
  - Code changes
  - Feature additions
- Helps ensure **software stability** after changes.

# 🧪 Types of Testing Involved:

| Type | Checks For |
|---|---|
| Resolution Testing | Verifies that a specific bug has been fixed |
| Regression Testing | Ensures other unaffected code still works |

**1. Redundant Test Cases**

•These are **duplicate or overlapping** tests that check the **same functionality** multiple times.

**2. Regression Test Cases**

•These are designed to **ensure that previously working functionality** still works correctly **after code changes, bug fixes, or enhancements**.

**3. Resolution Test Cases**

•These are the **newly added tests** to **verify that a specific bug fix (defect resolution)** has indeed corrected the problem.

**4. Invalid Test Cases**

•These tests are **obsolete, poorly designed, or no longer applicable** due to code or requirement changes.

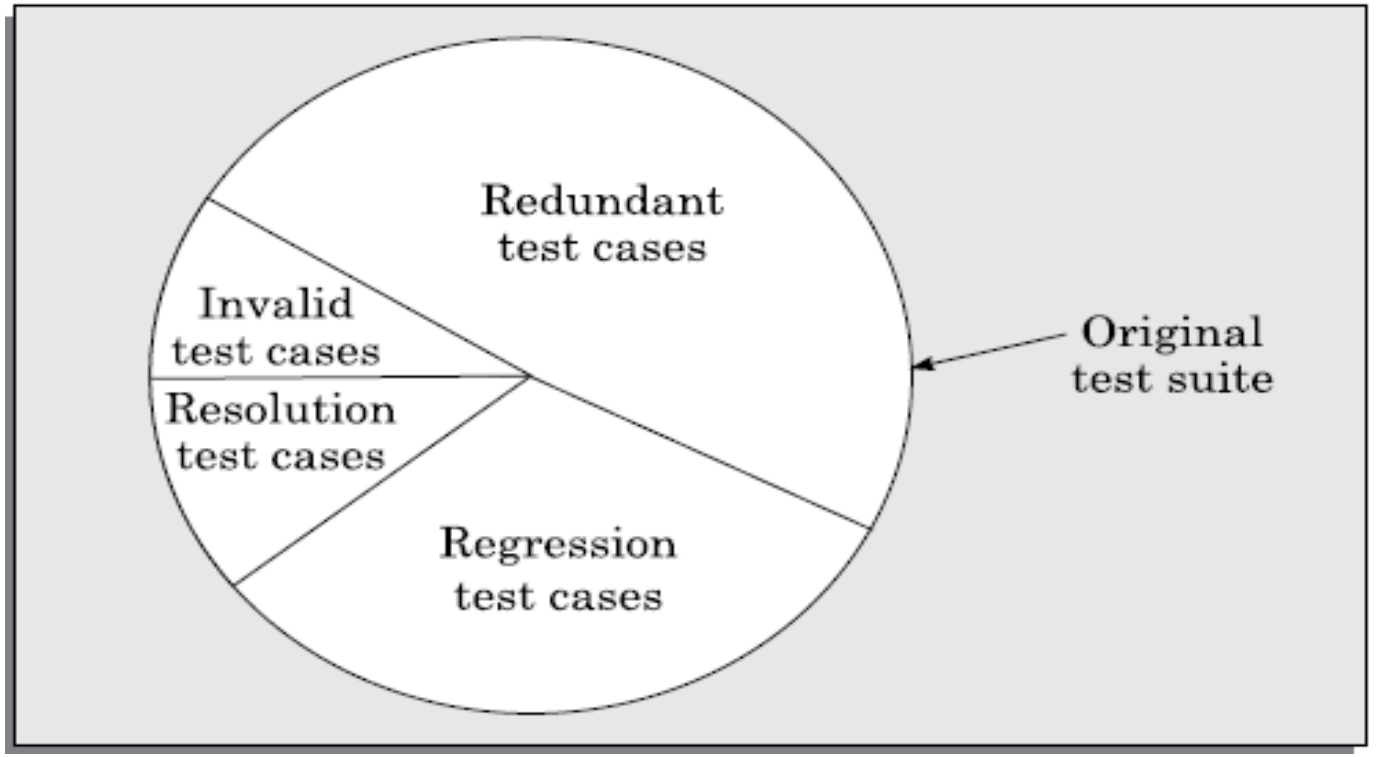



**FIGURE 7.9** Types of test cases in the original test suite after a change.

After a code change, the **original test suite** contains:

| Type of Test Case | Description |
|---|---|
| ✅ Valid Tests | Still applicable to the modified program |
| 🔁 Redundant Tests | Test unaffected parts of code (still correct but not essential to rerun) |
| ❌ Invalid Tests | No longer valid due to structural change in code or logic |
| 🎯 Affected Tests | Directly test the changed or impacted code (must be rerun) |

**Regression Testing Strategy:**

1. **Discard** invalid ones.
2. **Retain** unaffected ones as needed (for full assurance or periodic testing).
3. **Automate**: Automating test cases makes regression testing **quick and cost-effective**.

## 🚀 Benefits of Regression Testing:

- Maintains **software reliability**
- Prevents **reintroduction** of old bugs
- Reduces risk of **side effects** from changes
- Ensures **long-term code stability**

# Security Testing

**Security Testing** is a type of software testing that ensures that the system protects data and maintains functionality as intended. It is done to identify vulnerabilities and weaknesses in the software system that may be exploited by attackers or unauthorized users.

## 🔐 Objective of Security Testing:

- To **protect software systems** and data from unauthorized access, misuse, or damage.
- To ensure **confidentiality, integrity, authentication, authorization, availability**, and **non-repudiation**.

## 🧩 Key Aspects of Security Testing:

| Aspect | Description |
|---|---|
| 🔑 Authentication | Ensures that the system correctly verifies user identity (e.g., login security). |
| 🔒 Authorization | Ensures users can only access resources or perform actions they are permitted to. |
| 📦 Confidentiality | Ensures that sensitive data is accessible only to authorized users. |
| 🛠️ Integrity | Ensures data is not modified or tampered with without authorization. |
| 📡 Availability | Ensures that services remain available and accessible even under attack (e.g., DDoS). |
| 📃 Non-repudiation | Ensures that a user cannot deny the authenticity of their actions. |

## 🛡️ Common Types of Security Testing:

| Type | Purpose |
|---|---|
| Vulnerability Scanning | Automated scanning to find known vulnerabilities. |
| Penetration Testing (Pen Testing) | Simulated attack to find security weaknesses. |
| Security Auditing | Reviewing code and configurations for security compliance. |
| Ethical Hacking | Legally hacking the system to find vulnerabilities before attackers do. |
| Risk Assessment | Identifying potential security risks and impact. |
| Security Posture Assessment | Evaluating overall security and identifying gaps in protection. |

## 🧪 Steps in Security Testing:

1. **Understand security requirements** of the application.
2. **Identify potential threats** and vulnerabilities (e.g., SQL injection, XSS).
3. **Create threat models** or risk-based test cases.
4. **Execute security tests** (manual and automated).
5. **Analyze results**, log security defects.
6. **Fix vulnerabilities** and **retest** for resolution.
7. **Verify** against security standards (e.g., OWASP Top 10, ISO 27001).

## 🛠️ Security Testing Tools (Examples):

| Tool Name | Purpose |
|---|---|
| OWASP ZAP | Web application vulnerability scanning |
| Burp Suite | Penetration testing |
| Nessus | Vulnerability assessment |
| Metasploit | Exploitation framework |
| Fortify | Static code analysis |

## 🚨 Common Security Vulnerabilities:

- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Buffer Overflow
- Insecure Authentication
- Data Exposure
- Insecure APIs

## 🎯 Why Security Testing is Critical?

- Prevents **data breaches** and **financial losses**
- Protects **reputation** of the organization
- Ensures **legal and regulatory compliance**
- Builds **user trust** in the system

# 💪 Robustness Testing

**Robustness Testing** is a **type of software testing** that checks whether a software system behaves correctly and **gracefully handles unexpected or invalid inputs, stressful conditions, and system failures**.

## 🎯 Objective of Robustness Testing:

To ensure that the software:

- **Does not crash** or behave unpredictably when faced with **invalid**, **unexpected**, or **extreme input conditions**.
- Handles **stressful system states** (like memory shortages or hardware failures) **gracefully**.
- Provides **meaningful error messages** and **recovers properly** without loss of functionality.

## 🧪 What Robustness Testing

| Covers: Aspect | Example/Test Case |
|---|---|
| Invalid inputs | Entering special characters, extremely large numbers, etc. |
| Boundary conditions | Inputs near min/max limits of data types |
| Error handling | System behavior when a file is missing or connection fails |
| Hardware/software failures | Test behavior when disk is full, network is down, etc. |
| Incorrect API usage | Calling functions with wrong parameter types |

## 🔍 Difference Between Robustness and Reliability Testing:

| Aspect | Robustness Testing | Reliability Testing |
|---|---|---|
| Focus | Behavior under invalid/unexpected conditions | Behavior under normal, expected conditions |
| Objective | Ensure system doesn't break with bad input or failures | Ensure consistent and correct output over time |
| Input Type | Invalid, unexpected, or extreme inputs | Valid inputs only |

## 🛠️ Approaches in Robustness Testing:

**1.Fault Injection**

Simulate hardware faults, memory corruption, or network failure.

**2.Fuzz Testing (Fuzzing)**

Random data is input to the system to check for crashes or errors.

**3.Boundary Value Testing**

Inputs at the edge of allowable ranges are tested.

**4.Negative Testing**

Supplying invalid or wrong inputs to ensure the system can handle them.

## 🧰 Tools for Robustness Testing:

| Tool | Description |
|---|---|
| FuzzTester | Random input generation for fuzz testing |
| Jtest | Java-based robustness and exception testing |
| Ballista | Robustness testing of OS and libraries |
| Defensics | Protocol fuzzing and robustness testing |

## 🔄 Benefits of Robustness Testing:

- Prevents **system crashes and data corruption**
- Ensures **high availability** even in adverse conditions
- Enhances **user trust** by making the system resilient
- Prepares software for **real-world unpredictable scenarios**

✅

**Example:**

```
int divide(int a, int b) {
  return a / b;
}
```

**Robustness Testing Case:**

Test with b = 0 (division by zero) → Should gracefully handle this with an error message, not crash.

# 🧪 Fuzz Testing (Fuzzing)

**Fuzz Testing**, also known as **fuzzing**, is an automated software testing technique that involves **feeding random, unexpected, or malformed data** into a program to uncover **vulnerabilities**, **crashes**, and **unexpected behavior**.

## 🎯 Objective of Fuzz Testing:

- To detect **security loopholes**, **buffer overflows**, **input validation issues**, and **system crashes**.
- To test how well the software **handles invalid or unpredictable inputs**.

## ⚙️ How Fuzzing Works:

1. **Fuzz Input Generator** creates **random, invalid, or semi-valid inputs**.
2. These inputs are fed into the **System Under Test (SUT)**.
3. A **monitoring system** observes the program's behavior:
    1. Crashes
    2. Hangs
    3. Memory leaks
    4. Unexpected outputs
4. If abnormal behavior is detected, the test **logs the input** that caused it.

## 🔍 Types of Fuzz Testing:

| Type | Description |
|---|---|
| Mutation-based Fuzzing | Modifies existing valid inputs slightly to create new test cases. |
| Generation-based Fuzzing | Generates inputs from scratch based on input format rules. |
| Protocol-aware Fuzzing | Tests structured inputs for protocols (e.g., HTTP, FTP). |
| File format fuzzing | Sends corrupted images, audio, or files to test apps or systems. |

## 📌 Common Vulnerabilities Found via Fuzzing:

- Buffer overflows
- Memory leaks
- Stack overflows
- Input validation flaws
- SQL/Command injection
- Denial of Service (DoS)

## 🛠️ Popular Fuzzing Tools:

| Tool | Description |
|---|---|
| AFL (American Fuzzy Lop) | Powerful, open-source, coverage-guided fuzzer |
| LibFuzzer | In-process, coverage-based fuzzer for LLVM |
| Peach Fuzzer | Commercial tool supporting file and network fuzzing |
| OSS-Fuzz | Google's platform for fuzzing open-source software |
| Burp Suite | Web app fuzzing, especially for input validation |

## ✅ Example Scenario:

A login form expects:

Username: alphanumeric, max 10 chars
Password: alphanumeric, max 12 chars

## Fuzz Input Example:

- Username: @@@@@@@#####$$$$$%%%%%...
- Password: '; DROP TABLE users;--

If the software doesn't validate inputs properly, it might crash or even execute harmful code (like SQL injection).

## ⚖️ Advantages of Fuzz Testing:

- Detects **hidden bugs** not found in functional testing.
- Effective in uncovering **security vulnerabilities**.
- **Low cost**, can be largely **automated**.
- Good for **stress and robustness** testing.

## ⚠️ Limitations of Fuzz Testing:

- Doesn’t guarantee full test coverage.
- May miss **logical errors**.
- Hard to interpret if many false positives occur.
- May require significant **manual analysis** of crash logs.

## 🧠 Best Practices:

- Combine fuzzing with **static and dynamic analysis**.
- Use **instrumentation** to track code coverage.
- Always **log and replay** crashes to reproduce and fix bugs.

## 🧵 Fuzz Testing vs Robustness Testing:

| Aspect | Fuzz Testing | Robustness Testing |
|---|---|---|
| Input Strategy | Random or malformed inputs | Invalid or stressful inputs |
| Focus | Find vulnerabilities and crashes | Graceful handling of errors |
| Automation | Highly automated | Can be semi-automated or manual |
| Common Use | Security testing, bug hunting | Reliability and system stability |

# 🔗 Integration Testing

**Integration Testing** is a level of software testing where individual modules are **combined and tested as a group** to expose **interface defects** between them. It focuses on **data flow** between units and how well they **collaborate**.

## 🎯 Goal of Integration Testing:

- To verify that **modules or components interact correctly**.
- To identify **errors in interfaces**, **data sharing**, or **inter-module logic**.
- To ensure the **integrated system behaves as expected** before system testing.

## Why Integration Testing is Needed:

Even if each module works perfectly in isolation (after **unit testing**), **errors may arise** when modules:

- Exchange data.
- Use shared memory or global variables.
- Call one another with wrong parameters.
- Rely on timing or sequence.

## 🔍 Types of Integration Testing Approaches:

| Approach | Description |
|---|---|
| Big Bang | All modules are integrated and tested together. |
| Top-down | Starts with top-level modules and integrates downward using stubs. |
| Bottom-up | Begins with low-level modules and integrates upward using drivers. |
| Sandwich / Hybrid | Mix of top-down and bottom-up strategies. |
| Incremental | Modules are integrated and tested one by one in sequence. |

## 🔄 Terminology:

- **Stub**: Dummy module used to simulate lower-level modules (used in top-down testing).
- **Driver**: Dummy module used to simulate higher-level modules (used in bottom-up testing).

## 🧪 Integration Testing Example:

Consider 3 modules in an e-commerce system:

1. **Login Module**
2. **Product Catalog**
3. **Payment Gateway**

Unit testing ensures each works individually. In integration testing:

- Ensure after login, product catalog loads properly.
- Ensure selecting a product passes correct info. to the payment gateway.

## ✅ Advantages of Integration Testing:

- Catches interface issues early.
- Helps uncover design flaws.
- Ensures data communication is correct.
- Prevents bugs during system assembly.

## ⚠️ Challenges:

- Requires careful planning of module dependencies.
- Stub/driver creation may take effort.
- Interface errors can be harder to debug than isolated ones.
- Complex in large systems with many modules.

## 🧠 Best Practices:

- Plan integration order logically (least dependent modules first).
- Automate test scripts if possible.
- Test both **valid** and **invalid** data flows.
- Use continuous integration (CI) tools to catch issues early.
- Maintain proper logs to track issues between components.

## 🔁 Integration Testing vs Unit & System Testing:

| Type | Scope | Responsibility |
|---|---|---|
| Unit Testing | Individual functions/modules | Developers |
| Integration Testing | Combined modules & their interfaces | Developers/Testers |
| System Testing | Entire system against requirements | QA/Testers |

# Testing Object-Oriented Programs (OOP)

Testing object-oriented software differs from traditional procedural software due to **encapsulation**, **inheritance**, **polymorphism**, and **interaction between objects**. Let's explore how testing is approached in the context of object-oriented programming (OOP).

## 🔍 Key Challenges in OOP Testing:

| OOP Concept | Testing Challenge |
|---|---|
| Encapsulation | Data is hidden; internal state is hard to access directly for testing. |
| Inheritance | Behavior is inherited; base class changes can affect derived class behavior. |
| Polymorphism | A method call may refer to many actual implementations, making test prediction hard. |
| Dynamic Binding | Methods are resolved at runtime; exact behavior is known only during execution. |
| Object Interaction | Objects collaborate via messages; fault may lie in interactions rather than logic. |

## 🧰 Types of Tests in OOP:

### 1. Unit Testing

- Focuses on **individual classes and methods**.
- Verifies **method logic**, **constructors**, and **getter/setter behaviors**.
- Tools: JUnit (Java), pytest (Python), NUnit (.NET).

### 2. Integration Testing

- Tests **collaboration** among objects.
- Verifies **message passing**, **shared data**, and **interface contracts**.
- Mocks/stubs often simulate object dependencies.

### 3. System Testing

- Entire system is tested as a whole to ensure it meets requirements.
- Object interactions, flows, and business logic are verified in a full environment.

## 🧪 OOP-Specific Testing Strategies:

### ✅ Method Testing

- Test all public methods.
- Include tests for overloaded methods and methods with default arguments.

### 🔁 State-Based Testing

- Focuses on class behavior **based on its internal state**.
- Ensures correct transitions between states.

### 📥 Interaction Testing

- Ensures that **messages passed between objects** follow the expected sequence and correctness.
- Often tested using **mocking frameworks**.

### 🧬 Inheritance & Polymorphism Testing

- Test both **base** and **derived** class behavior.
- Ensure that overridden methods maintain behavioral contracts.
- Use **Liskov Substitution Principle (LSP)** to ensure objects of derived types can replace base types without altering functionality.

## 🧠 Best Practices for Testing OOP:

1. **Design for testability** – make classes loosely coupled and use interfaces.
2. **Keep classes small** – single responsibility principle helps in easier testing.
3. **Avoid tight coupling** – prefer dependency injection.
4. **Use mocks and stubs** – simulate collaborators in tests.
5. **Ensure 100% code coverage** for polymorphic and inherited methods.

## 🧪 Tools for Testing OOP:

| Language | Framework |
|---|---|
| Java | JUnit, TestNG |
| Python | unittest, pytest, mock |
| C++ | Google Test (gtest) |
| C#/.NET | NUnit, xUnit |
| JavaScript | Jest, Mocha |

# SYSTEM TESTING

## ✅ Definition:

System testing validates a **fully integrated software system** to ensure it meets all **functional and non-functional requirements** outlined in the SRS document.

## 🧪 Types of System Testing:

These differ based on **who performs** the test:

| Type | Description |
|---|---|
| Alpha Testing | Conducted by developers' internal team before release. |
| Beta Testing | Performed by select external customers in a real-world setting. |
| Acceptance Testing | Done by the customer/client to decide whether to accept the software. |

## Smoke Testing:

- **Purpose**: To verify that the **main functionalities** of the system work before full system testing begins.
- **Example**: For a library system, smoke tests may check book loan/return and user registration.
- Also known as **build verification testing**.

## ⚙️ Performance Testing:

Tests whether the system meets its **non-functional requirements**.

- **Types of Performance Testing:**

| Test Type | Purpose & Example |
|---|---|
| Stress Testing | Push the system beyond its design limits (e.g., more users or transactions). |
| Volume Testing | Examine performance with large data volumes (e.g., compiler with 100,000 lines). |
| Configuration Testing | Check system across different hardware/software configurations. |
| Compatibility Testing | Ensure correct interfacing with external systems like databases or networks. |
| Recovery Testing | Simulate failures (e.g., power off, device disconnect) and check recovery. |
| Maintenance Testing | Test diagnostic tools or update mechanisms for system maintainability. |
| Documentation Testing | Validate manuals and instructions for users and maintainers. |
| Usability Testing | Test the user interface (UI) for compliance with user-friendliness. |
| Security Testing | Verify robustness against threats (e.g., penetration testing, password attacks). |

## Error Seeding:

- **Purpose**: Estimate the number of **residual (undetected) errors** and evaluate the **effectiveness** of the test suite.
- **Process**:
  - Seed known artificial errors (S).
  - Count how many of them are found during testing (s).
  - Count actual (non-seeded) errors found (n).

**Estimating Total Errors (N):**

$$\frac{n}{s} = \frac{N}{S} \Rightarrow N = \frac{S \cdot n}{s}$$

**Estimating Remaining Errors:**

$$N - n = \frac{S \cdot n}{s} - n = n \cdot \left( \frac{S - s}{s} \right)$$

**Note: Accuracy depends on how closely seeded errors reflect real ones.**

# GENERAL ISSUES IN TESTING

## 📝 1. Test Documentation

The **Test Summary Report** is prepared at the end of testing and includes:

- ✅ Total number of tests conducted
- ✅ Number of successful tests
- ❌ Number of unsuccessful tests
- Degree of failure (partial or complete)

## 🔁 2. Regression Testing

**Definition**: Running old test cases **after any code change or bug fix** to ensure no new bugs are introduced.

- It spans across **unit, integration, and system testing**.
- ✅ Run only those test cases that could be affected by the change.
- 📌 Difference from Resolution Testing:
  - **Resolution Testing**: Checks if a defect has been fixed.
  - **Regression Testing**: Checks that previously working features **still work**.
- 🔁 Automated regression testing can save **significant effort**, especially as some tests may become:
- 🔸 **Invalid** (due to change),
- 🔹 **Redundant** (not affected by the change),
- ⚠️ Or may need to be added for new functionality.

## 🤖 3. Test Automation

Used to **reduce human effort, cost, and time**, while **increasing accuracy and test coverage**.

### 🔹 Benefits:

- ➢ Eliminates monotony and human error in repeated testing
- ➢ Supports thorough testing with large test suites
- ➢ Essential for **regression testing**, especially after frequent changes

### 🔸 Types of Test Automation Tools:

| Tool Type | Description |
|---|---|
| Capture & Playback | Records input-output during manual testing, then replays them automatically |
| Test Script-Based | Uses scripts (in various languages) to automate test execution and validation |
| Random Input Test | Randomly generates inputs to crash the system; doesn't check output correctness |
| Model-Based Test | Uses models (e.g., state diagrams, activity diagrams) to generate test cases |

- **Capture & Playback** tools are great for **regression testing**, but maintaining tests after changes can be expensive.
- **Test Script** tools require effort for debugging and updating after any code change.
- **Random Testing** is limited—it finds **crash-causing** defects only.
- **Model-Based Testing** can be powerful for validating system behavior.