# CS3343 notes

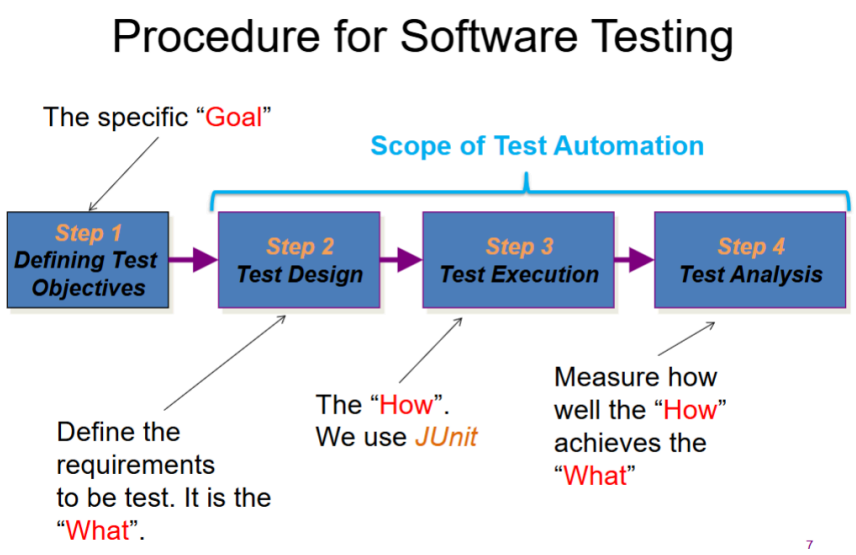
## Software Testing

- involves executing a program with input data to check outputs and identify problems
- **Key Limitation**: Testing can only demonstrate the presence of errors, not prove their absence.

### Purpose

- Find as many bugs (faults) as possible at the lowest cost.
- Balance effectiveness (comprehensive test coverage) and efficiency (minimal time).

### Procedure




- **Step 1**: Defining Test Objectives
  - Define the justification, purpose, or goal for testing
- **Step 2**: Test Design
  - Write test specifications or test code (e.g., JUnit scripts) to meet the objectives defined in Step 1.
- **Step 3**: Test Execution
  - Run the test cases designed in Step 2, better to automate nightly builds and tests
- **Step 4**: Test Analysis
  - Identify failures from test outputs (e.g. use 'assertEquals' in JUnit )

# JUnit Framework

## assertEquals Function

- Formula: assertEquals(expected, actual)
- Compares two objects for equality; throws an AssertionError if they differ.

## Operating JUnit with Eclipse

1. Create a new Eclipse project.
2. Create separate packages for the program under test and test cases.
3. Run the program to verify functionality.
4. Create JUnit test case classes.
5. Run the test driver.
6. (Optional) Create and run a JUnit test suite.

## Example for Test Automation using JUnit

Here is the Java implementation of a Poker game to check if a hand of 5 cards is a full house:

```
package poker;

public class Poker {
   // Precondition: A hand of n cards have been sorted by card number.
   public boolean isFullHouse(String cards[], int n) {
      return isThreeOfaKind(cards, n) && isTwoPairs(cards, n);
   }

   // Precondition: A hand of n cards have been sorted by card number.
   public boolean isThreeOfaKind(String cards[], int n) {
   for (int i = 0; i < n - 2; i++) {
     if (cards[i].charAt(1) == cards[i + 1].charAt(1) &&
          cards[i + 1].charAt(1) == cards[i + 2].charAt(1))
        return true;
   }
   return false;
   }

   // Precondition: A hand of n cards have been sorted by card number.
```

```
    public boolean isTwoPairs(String cards[], int n) {
    int count = 0;
    for (int i = 0; i < n - 1; i++) {
      if (cards[i].charAt(1) == cards[i + 1].charAt(1)) {
          count++;
          i++;
      }
    }
    if (count == 2)
      return true;
    else
      return false;
    }

    public static void main(String args[]) {
        System.out.println(new Poker().isFullHouse(new String[] { "C2", "D2", "H2", "S
3", "S4" }, 5));
    }
}
```

The code below demonstrates how to write JUnit test cases to automate testing of the Poker class.

```
package testPoker;
import poker.Poker;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class PokerTest {
    private Poker poker;
    /**
 * Sets up the test fixture.
 * Called before every test case method.
 */
    @BeforeEach
    public void setup() throws Exception {
        poker = new Poker();
    }

    // Test case 1: n = 0, cards = {}
    @Test
    public void testNoCards() {
```

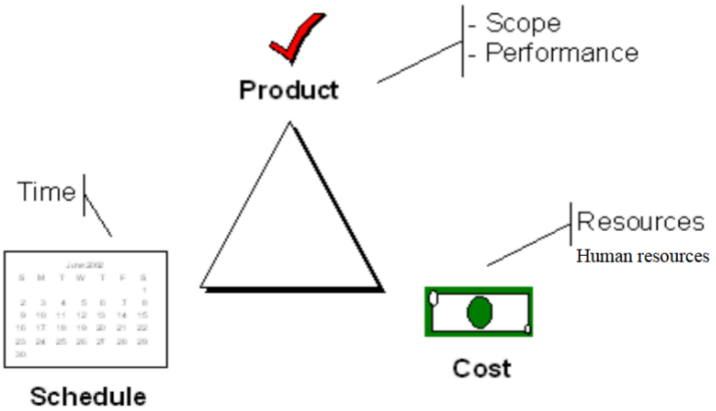
```
    boolean result;
    String[] input = null;
    result = poker.isFullHouse(input, 0);
    assertEquals(false, result);
    }

    // Test case 2: n = 5, cards = {"C2", "D2", "H2", "S3", "S4"};
    @Test
    public void test22234() {
    boolean result;
    String[] input = new String[]{"C2", "D2", "H2", "S3", "S4"};
    result = poker.isFullHouse(input, 5);
    assertEquals(false, result);
    }
}
```

# Software Project Management (SPM)

## PM Function (Iterative Process)

- Steps (Repeat until project ends.):
  1. Plan
  2. Schedule
  3. Track
  4. Measure

## Mini Trade-off Triangle




- Constraints: Fast, Cheap, Good (can only choose two).

## Successful Project Criteria

- Completed within time and budget.
- Customer requirements satisfied/exceeded.
- Accepted by the customer.

## Key Reasons for Project Failures

- Keeps changing project scope
- Poor requirements
- Unrealistic / No planning and scheduling
- Lack of resources/skills/management support
- Ineffective team

## 4 Project Dimensions

- People: Assign right tasks.
- Development Process: Follow structured process (e.g., testing).
- Product: Manage code/docs per requirements.
- Technology: Use right tools/patterns.

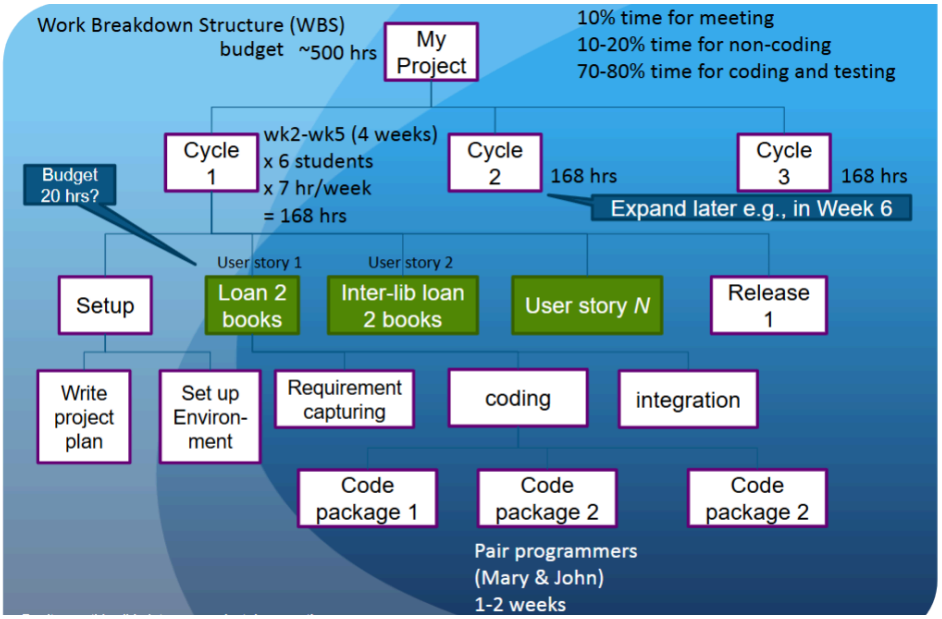
## Essential PM Strategies

- Handle 4 Project Dimensions
- Select proper development model (e.g., test-driven).
- Manage risks: **Avoid Mistakes**
    - People-Related Mistakes
    - Process-Related Mistakes
    - Product-Related Mistakes
    - Technology-Related Mistakes
    - Organizational-Related Mistakes, under:
        - Functional Organizational Structures (Specialized departments)
            - pros: clarity; cons: slow decisions.
        - Project-based Organizational Structures
            - pros: unity; cons: duplication.
        - Matrix Organizational Structures
            - pros: efficiency; cons: complexity.

- Follow software design principles/patterns.

# Project Planning and Scheduling

## Work Breakdown Structure (WBS)




- **Definition**: A hierarchical list of project work activities used to break down a project into manageable tasks.
- **Purpose**: Organizes tasks to facilitate planning, scheduling, and resource allocation.
- **Formats**:
  - Outline (indented format with decimal numbering, e.g., 3.1.5 for three levels).
  - Graphical Tree (organizational chart).

## Scheduling Process

- Steps:
  1. **Identify Tasks (What)**
     - Use WBS to define tasks
  2. **Estimate Software Size (How Much)**
     - Techniques include Lines of Code, Functions, or Requirements
  3. **Identify Task Dependencies**
     - Create a dependency graph or network diagram
     - Minimize dependencies to avoid delays

4. **Estimate Total Duration**
    - Develop the actual schedule
    - Allocate tasks to resources

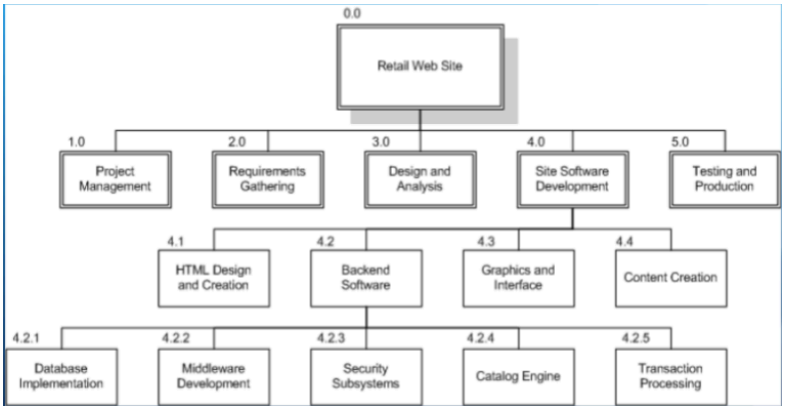
- **Primary Objectives**:
    - Minimize time (best time)
    - Minimize cost (least cost)
    - Minimize risk (least risk)
- **Secondary Objectives**:
    - Evaluate schedule alternatives
    - Ensure effective resource use
    - Facilitate communication

## Techniques to Produce WBS

- **Top-Down**:
    - Start at the highest level and fill in details.
- **Bottom-Up**:
    - Start with detailed tasks and consolidate into higher levels.
- **Analogy**:
    - Use similar projects as a template and modify as needed.
- **Brainstorming**:
    - Brainstorm all possible activities and group them into categories (e.g., using post-its).

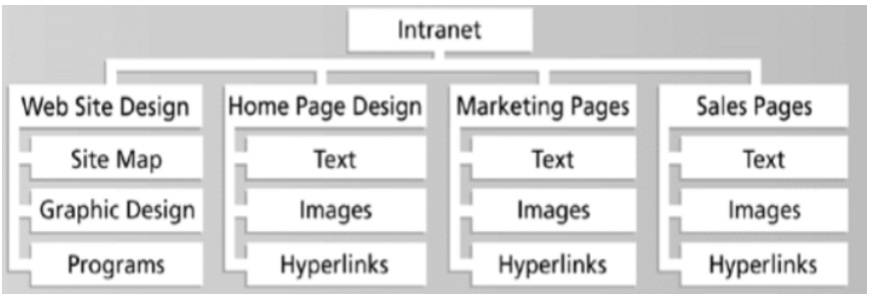
## Types of WBS

- **Task-Driven WBS**:



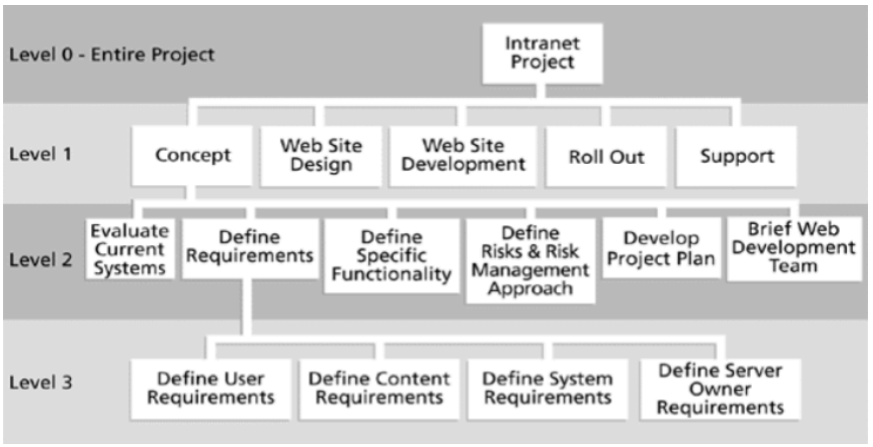
- Focuses on activities (e.g., development, testing).

- **Product-Type WBS**:




  - Organized around deliverables or components (e.g., website design, database).

- **Process-Type WBS**:



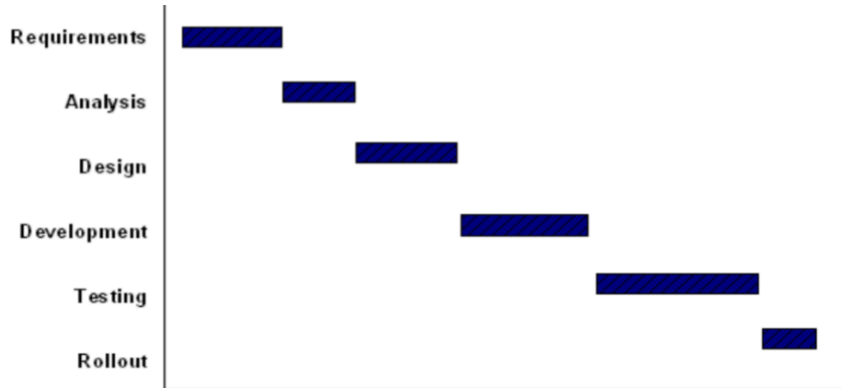

  - Structured around project phases (e.g., concept, design, development, rollout).

## Bottom-Level Tasks (WBS Leaves)

- **Definition**: Discrete tasks with definable end results, typically the lowest level of the WBS ("leaves").
- **Characteristics**:
  - Follow the "one-to-two" rule: Tasks assigned to 1–2 persons for 1–2 weeks.
  - Basis for monitoring and reporting progress.
  - Tied to budget items (charge numbers) and resource assignments.
  - Ideal duration: 1 day (minimum, occasionally half-day) to 2–3 weeks (maximum for software projects).
  - Longer tasks require in-progress estimates, which are subjective.
  - Avoid overly small tasks to prevent micromanagement.

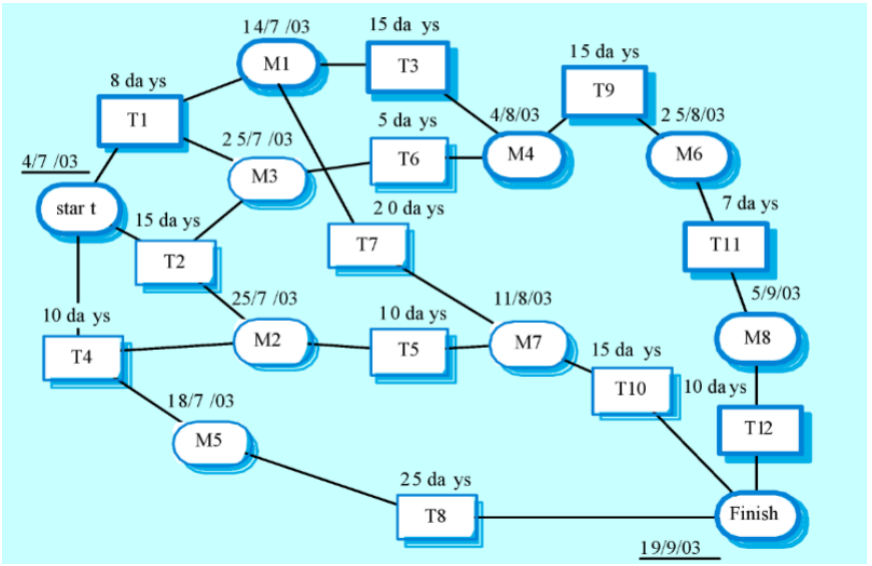
## Scheduling Tools

- **Bar Charts (Gantt Charts)**:



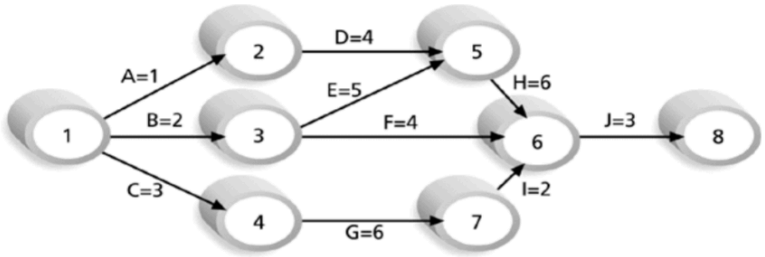

  - Illustrate project schedules against calendar time.
  - Show task breakdowns and durations.

- **Activity Networks**:




  - Depict task dependencies and critical paths.
  - Used to analyze the sequence and relationships between tasks.

## Critical Path Method (CPM)




Note: Assume all durations are in days.

| | | |
|---|---|---|
| Path 1: | A-D-H-J | Length = 1+4+6+3 = 14 days |
| **Path 2:** | **B-E-H-J** | **Length = 2+5+6+3 = 16 days** |
| Path 3: | B-F-J | Length = 2+4+3 = 9 days |
| Path 4: | C-G-I-J | Length = 3+6+2+3 = 14 days |

**Since the critical path is the longest path through the network diagram, Path 2, B-E-H-J, is the critical path for Project X.**

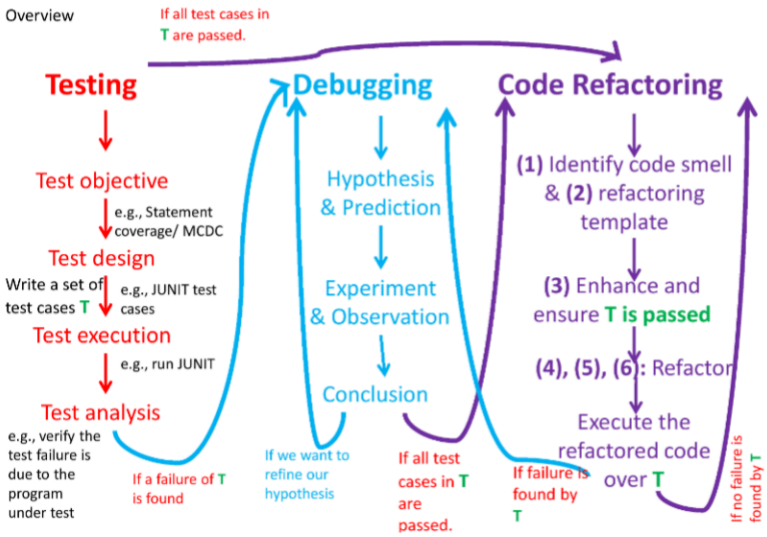
- **Definition**: Identifies the specific set of sequential tasks that determine the project's completion date (the longest full path).
- **Formula for Critical Path Calculation**:
  - For a path (e.g., A-D-H-J): *Length = A + D + H + J*.
  - The critical path is the path with the longest total duration.
- **Key Points**:
  - All projects have a critical path.
  - Accelerating non-critical tasks does not directly shorten the project duration.

### Event Times in Activity Networks

- **Earliest Event Time (EET)**:
  - The earliest time an event can occur, calculated when all preceding activities are complete.
  - Determined by working forward through the network from start to finish.
- **Latest Event Time (LET)**:
  - The latest time an event can occur without delaying subsequent events.
  - Determined by working backward from the finish node to the start node.
- **Critical Activities**:
  - Activities that cannot be delayed without delaying the project (lie on the critical path).

## Testing-Debugging-Refactoring

### Cycle




- **Testing**: Define objectives, design/execute tests (e.g., JUnit), analyze failures.
- **Debugging**: Fix bugs if tests fail unexpectedly.
- **Refactoring**: Identify code smells, apply templates, ensure tests pass.
- **Goal**: Pass all tests, then improve code structure.

## Code Smell

- **Definition**: A symptom in the source code that may indicate a deeper problem (e.g., poor design or implementation).
- **Purpose**: Identifying code smells guides refactoring to improve code quality.

## Code Refactoring

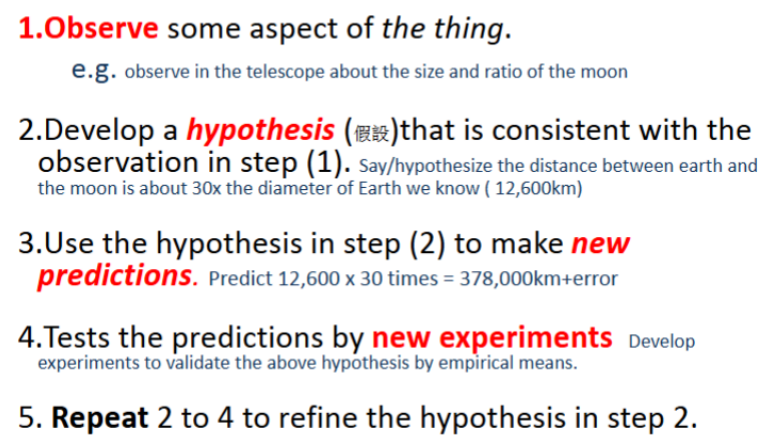
- **Purpose**:
  - Improve design incrementally (counteract decay from quick fixes or over-design).
  - Enhance maintainability/extensibility (supports Open-Closed Principle).
  - Speed up development via modular, clear code.
- **When to Refactor**:
  - Repetitive code patterns (e.g., third copy-paste).
  - Before adding functionality or fixing bugs.
  - During code reviews to address code smells.
- **When Not to Refactor**:
  - During new functionality/bug fixes.
  - Without unit tests, non-functional code, or near deadlines.

- **Collapse Hierarchy Template**:
  - Merge similar superclass/subclass.
  - Steps: Remove subclass, pull up methods/fields, recompile/test, adjust references, remove subclass, retest.

### Debugging Mindset

- Strategic: Form hypothesis, predict outcomes, test, observe, conclude.
- Non-strategic: Vague assumptions lead to inconclusive results.
- Focus on requirements to develop targeted test cases.

### Debugging Technique




## Bug Reporting

### Bug Tracking Systems

- **Purpose**: Tools to manage and track bugs during software development.
- **Examples of Systems**: Bugzilla, Mantis, Trac.

### Bug Report Components

- **Essential Elements**:
  - **Situation**: Specify the context (e.g., OS, component, platform).
  - **Bug Description**: Detailed explanation of the issue.
  - **Bug Report Title**: Concise summary (e.g., 60 or fewer characters).
  - **Severity**: Impact level (e.g., minor, normal, major, critical; varies by organization).
  - **Steps to Reproduce**: Minimal steps to trigger the bug, including setup.

- **Expected Result**: What the program should do without the bug.
- **Actual Result**: Observed behavior, preferably with evidence (e.g., screenshot, stack trace).
- **Additional Details**:
  - **Product**: The software where the bug was found.
  - **Component**: Specific module or feature affected.
  - **Platform**: Hardware platform (e.g., PC, Macintosh).
  - **OS**: Operating system (e.g., Windows, Linux, Mac OS X).
  - **URL**: Relevant URL or HTML snippet if applicable.
  - **Assigned To**: Engineer responsible for fixing, assigned by the project manager.

### Qualities of a Good Bug Report

- **Reproducible**: Must include clear **Steps to Reproduce (STR)** with all relevant details to ensure the bug can be replicated.
- **Specificity**: Isolate the issue precisely to expedite fixing; vague reports become irrelevant with software changes.
- **Avoid Poor Practices**: Vague titles, unrelated issues, or emotional language reduce report effectiveness.

## Control-flow & Predicate Testing

| Type | Name | After running a set of test cases, |
|---|---|---|
| Control-flow | Statement Coverage | Each **statement has been executed at least once**. |
| | Branch Coverage | Each **branch has been executed at least once**. |
| | Loop coverage | Each **loop** should be tested with 0, 1, and >1 times<br>Each **branch has been executed at least once**. |
| | Path Coverage | Each **path** has been executed at least once. |
| Predicate | Condition Coverage (CC) | Each **condition** has been evaluated to be true and false. |
| | Decision Coverage (DC) | Each **decision** has been evaluated to be true and false. |
| | C/DC | Both **CC and DC** are satisfied by the same test set. |
| | MC/DC | 1. Both CC and DC are satisfied, and<br>2. Each conditional is shown to independently affect the outcomes of each decision. |

## Test Organization

### Testing Levels

- **Unit Test**:

    - **Description**: Tests individual classes in isolation.
    - **Focus**: Determine if a class is buggy (without calling other methods or classes).
    - **Performed By**: Developers.
- **Integration Test**:
    - **Description**: Tests interactions among a subset of classes (components) or subsystems (sets of components).
    - **Focus**: Identify bugs in interactions between classes or subsystems.
    - **Performed By**: Testers or independent testing teams.
- **System Test**:
    - **Description**: Tests the integration of all subsystems to form a complete system.
    - **Focus**: Verify if the system as a whole is faulty.
    - **Performed By**: Testers or independent testing teams.

## White-Box vs. Black-Box Testing

- **White-Box Testing**: use source code to design and verify test cases (e.g., execute all statements).
- **Black-Box Testing**: rely on specifications without assuming source code access (e.g., testing against use case scenarios).

## Integration Testing Strategies

- **Bottom-Up**:
    - **Description**: Tests lower-level modules first, progressing to higher-level modules.
    - **Dependencies**: Lower modules must be tested before higher ones.
- **Top-Down**:
    - **Description**: Tests higher-level modules first, progressing to lower-level modules.
    - **Dependencies**: Higher modules tested with stubs for lower modules.
- **Sandwich Testing**:
    - **Description**: Combines top-down and bottom-up approaches, testing some modules top-down and others bottom-up.
    - **Characteristics**:
        - Requires both test drivers and stubs.
    - **Popularity**: One of the most widely used strategies.

- **Modified Top-Down**:
  - **Description**: Tests higher-level modules first but with modifications to reduce stub complexity.
  - **Characteristics**:
    - Requires both test drivers and stubs.
- **Big-Bang**:
  - **Description**: Tests all modules simultaneously without intermediate steps.
  - **Characteristics**:
    - Requires many test drivers and stubs.
    - Hard to identify integration bugs.
    - Medium parallelism, hard to plan and control.
  - **Recommendation**: Undesirable and not used in modern development projects.