Coding Standards

Why Coding Standards are Important:

- Every developer writes code differently, which can cause problems when reusing, maintaining, or reviewing code.
- Having a common coding standard for all teams reduces these problems.
- Good coding standards focus on: modularity, clarity, simplicity, reliability, safety, and maintainability.

Modularity:

- Code should be divided into modules.
- Each important function should be inside its own module.
- Modules should handle both structure and data.
- If a feature is needed, it can be added using that specific module.
- Helps in reusing code, improves productivity, and makes code easier to read.

Clarity:

- Code should be easy to understand for anyone reading it.
- Use standard naming rules for variables, functions, and classes.
- Add enough comments to explain what each part of the code does.
- Use enough white spaces to make the code more readable.

Simplicity:

- Code should be simple and not have unnecessary complex logic.
- Simpler code is easier to read and fix.
- Follow best practices for simplicity, like using clean programming styles.
- Break big problems into smaller, meaningful parts.
- Use techniques like abstraction and information hiding in object-oriented programming.

Reliability:

- Code must be reliable, especially for end users.
- Follow standard software development processes.
- Clean and simple code makes it easier to find and fix mistakes.
- Plan for future updates while designing the structure.
- Avoid adding new code on a weak or bad structure.
- If needed, redesign the structure to prevent messy, tangled code ("spaghetti code").

Safety:

- Software in critical fields like healthcare or road safety must be extremely safe.

- Errors in such software can be life-threatening.
- The chance of errors should be very close to zero (less than 0.00001%).
- Code should have built-in safety checks and protections.

Maintainability:

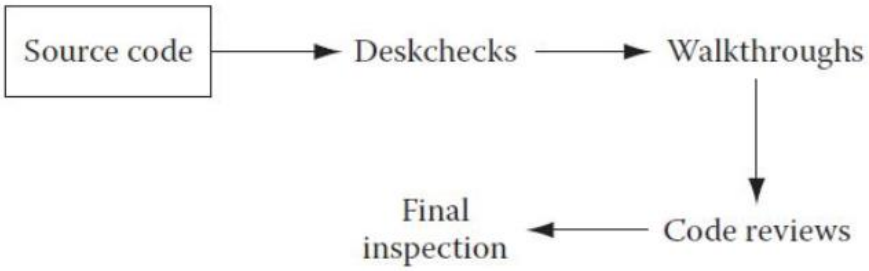
- Maintenance work uses more than 70% of the total software budget.
- Write code in a way that is easy to maintain.
- Maintainable code makes it easier and cheaper to fix problems during maintenance.

Coding Framework

- A coding framework gives a common structure for writing consistent, standard, and testable code.
- In object-oriented programming, base classes are defined for use throughout the project.
- Frameworks act like a strong foundation, allowing developers to build and extend easily.
- It improves developer productivity and ensures the product is strong and well-organized, just like building a house on a solid base.

Reviews (Quality Control)

- Around 70% of software defects happen because of bad code.
- Building software takes a lot of effort and time.
- If defects are found later, a lot of effort gets wasted.
- It is cheaper to fix defects during construction rather than during testing.
- Code reviews like deskchecks, walkthroughs, and inspections are very important to ensure good quality and to save costs.
- Different types of reviews happen at different stages for different purposes.
- Inspections are final reviews to decide if the code can move forward.
- Other reviews mainly focus on finding and fixing defects.



1. **Deskchecks (Peer Reviews)**

- Used when a full code review is not needed.
- Developer shows their code to team members for feedback.
- Team members give feedback voluntarily.

- Developer can choose to accept or ignore suggestions.
- Helps in finding and fixing defects and improving code quality.

2. **Walkthroughs**

- Developer asks for a formal but voluntary review.
- Developer explains the code, the logic, and the method used to the team.
- Team gives suggestions for betterment.
- Developer decides whether to apply or reject the suggestions.
- Very useful for finding defects early and improving code quality.

3. **Code Reviews**

- A formal review started by the project manager.
- The team reviews the developer's code to find mistakes, defects, or bad logic.
- An error log is created and discussed with the whole team.

4. **Inspections**

- Final and formal review of the code.
- It decides whether the code is ready to be added to the main software build.
- If code passes inspection, it is accepted; if not, it must be corrected.

Coding Methods

- Turning design into efficient software is very important.
- As hardware improved, software became bigger and more complex.
- Different coding methods were created to handle larger software needs.
- Advancements in computer science help in improving coding methods.

Main Coding Methods:

1. **Structured Programming**
    - Came with powerful mainframe computers.
    - Organizes code into small parts using functions and procedures.
    - Makes code reusable, readable, and easy to maintain.
2. **Object-Oriented Programming (OOP)**
    - Combines data and actions together.
    - Models real-world objects (like a car with parts and features).
    - Uses encapsulation to hide unnecessary details.
    - Makes systems more robust, modular, and easy to understand.
3. **Automatic Code Generation**
    - Coding manually takes a lot of time and effort.
    - Tools like CASE and ERP platforms can generate code automatically but are still basic.
    - Full automation of coding is still a challenge.
4. **Pair Programming**

    - Two developers work together: one writes code, the other reviews and guides.
    - They switch roles often.
    - Improves code quality, reduces mistakes, and helps share knowledge.
5. **Test-Driven Development (TDD)**
    - Write test cases before writing the actual code.
    - Helps make sure the logic is correct before coding.
    - Used mainly in iterative development like eXtreme Programming (XP).

Software Code Reuse

- **Why?** To save time and avoid rewriting common code.
- **How?**
    - In procedural programming: Use functions and libraries.
    - In OOP: Use inheritance, polymorphism, and encapsulation.
- **Benefits:** Saves time, improves quality, and ensures consistency.

Reuse Methods:

- **Libraries:** Collections of reusable code (functions, classes).
- **Open Source:** Reusing publicly available code shared by the community.
- **Software as a Service (SaaS):** Using cloud-based software instead of writing everything from scratch.
- **Inheritance:** Reusing and extending existing classes in OOP.

Strategic Approach to Software Testing

Key Characteristics:

- Conduct **formal technical reviews** to catch errors before actual testing.
- Start testing **at the component level**, then move to **system-level** testing.
- Use **different testing techniques** at each stage.
- Testing is done by **developers** and also by an **independent test group** (for large projects).
- **Testing** = finding defects.
  Debugging = fixing defects.
  (Both must be planned separately.)

Verification and Validation (V&V)

1.Verification

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Methods of Verification: Static Testing, Walkthrough, Inspection, Review

2.Validation

Validation is the process of evaluating a software system or component during, or at the end of, the development cycle to determine whether it satisfies specified requirements.

Methods of Validation: Dynamic Testing, Testing

Relation to SQA (Software Quality Assurance):

- V&V activities are part of SQA practices, including:
  - Formal reviews
  - Audits
  - Monitoring and simulations
  - Documentation reviews
  - Algorithm and database analysis
  - Different types of testing

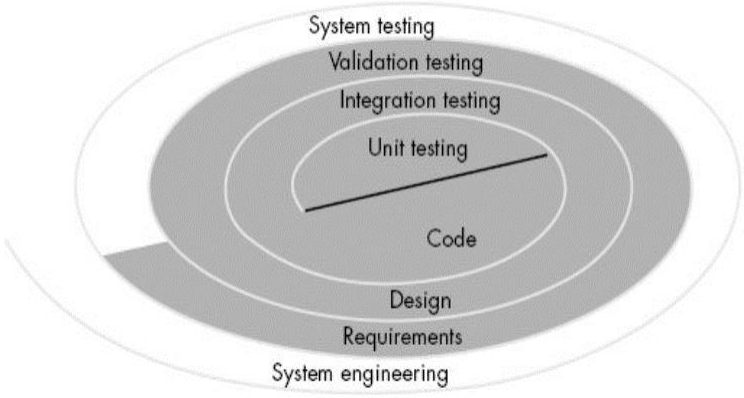
Quality is ensured by:

- Good application methods
- Proper process models
- Tools and thorough reviews
- Rigorous testing practices

Verification	Validation
Static practice (verifying documents, design, code).	Dynamic process (testing the actual product).
Does not involve executing code.	Always involves executing code.
Human-based checking (reviews, walkthroughs).	Computer-based execution (functional testing).
Methods: Inspections, Desk-checking, Walkthroughs.	Methods: Black box, Gray box, White box testing.
Ensures conformity to specifications.	Ensures meeting customer expectations.
Catches different kinds of errors; low-level exercise.	Catches other errors; high-level exercise.
Target: Requirements specs, design documents.	Target: Actual working product/modules.
Done mainly by QA team.	Done mainly by Testing team.
Performed before validation.	Performed after verification.

Software Testing Strategy: Spiral Approach

Testing moves outward like a spiral for thorough validation at every level.



Phase	Purpose	Techniques
Unit Testing	Test individual components/modules.	White-box testing (control paths, edge cases)
Integration Testing	Verify interactions between modules.	Black-box testing (module interface issues)
Validation Testing	Ensure system meets functional & performance requirements.	Black-box testing
System Testing	Full system integration and performance check.	Combination of testing techniques

Criteria for Completion of Testing

- **Testing is never completely finished.**
- Completion is based on:
  - **Time:** Deadline is reached.
  - **Budget:** Testing resources are exhausted.
- **Goal:** Minimize defects within limits.
- Even users unknowingly test the system every time they use it!

1. Unit Testing

- **Focus:** Verifies the smallest design unit (component/module).
- **Approach:** Tests critical control paths and finds errors inside module boundaries using component-level design.
- **Orientation:** White-box testing, often conducted in parallel for multiple components.
- **Key Aspects:**
  - Identify inputs, outputs, and edge cases.
  - Define systematic steps to execute and evaluate tests.

Criteria for Completion of Testing

- Absolute failure-free software **cannot be guaranteed**.
- Testing should be based on **statistical criteria**.
- Example: **95% confidence** level achieved through validated models.
- Reliability grounded in **theoretical and experimental standards**.

Key Unit Testing Focus Areas

Area	Details
Module Interface	Verify information flow into and out of the unit first.
Local Data Structure	Ensure integrity of temporary data storage.
Boundary Conditions	Test module functionality at its input and output limits.
Independent Paths	Execute every statement at least once.
Error Handling Paths	Confirm robustness through exception paths.

Module Interfaces, Local/Global Data, and Selective Testing

- **Module Interfaces:** Must be verified first — wrong data flow ruins all other tests.
- **Local and Global Data:** Test their behavior and impact.
- **Selective Testing Targets:**
  - **Computations** (arithmetic, operations)
  - **Comparisons** (logical operators, precedence)
  - **Control Flow** (loops, conditions)
- **Techniques:**
  - **Basis Path Testing:** Identify errors in logic paths.
  - **Loop Testing:** Detect loop-related errors.

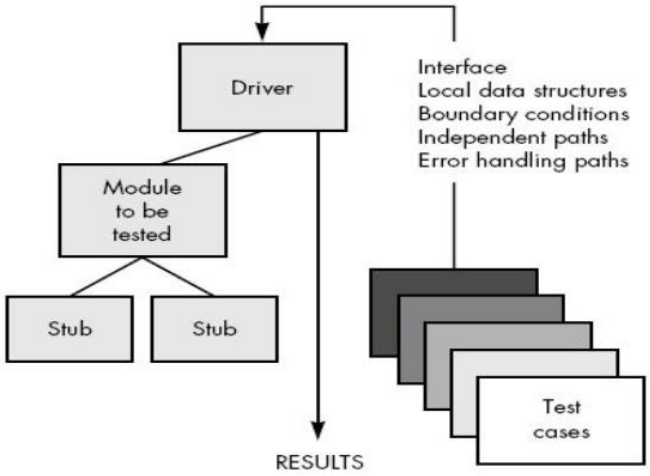
More Unit Testing Strategies:

- **Evaluate Error Descriptions:** They must be clear and help locate the root cause.
- **Test Exception Handling:** Verify no unintended system behavior before proper error handling.
- **Boundary Value Analysis (BVA):**
  - Most errors occur at boundaries (min, max, edge values).
  - Design test cases just **below, at, and above** the boundaries.
  - **BVA should be the final step** in unit testing.

Unit Test Procedures

- **Timing:** Done either **before coding** or **after source code generation**.
- **Test Case Design:** Based on design review, each test case should define expected results.

- **Drivers:** Temporary programs that:
  - Simulate the "main" program.
  - Accept test data, pass it to the component, and display results.
- **Stubs:** Temporary modules that:
  - Mimic subordinate modules.
  - Perform minimal operations and return control.



Common Errors in Unit Testing:

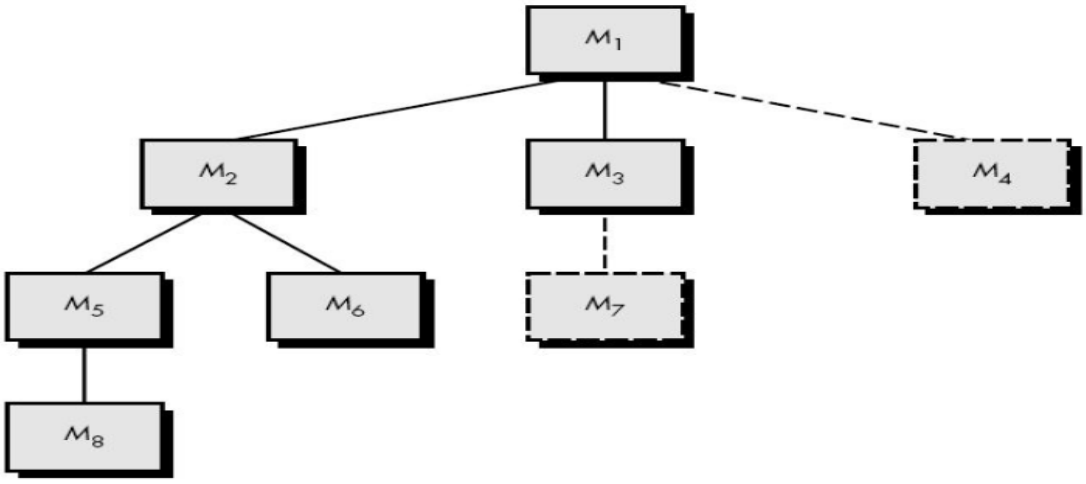
Area	Common Issues
Computation	Incorrect arithmetic precedence, mixed operations, wrong initialization, precision issues, wrong symbolic expressions.
Comparison and Control Flow	Data type mismatch, wrong logical operators, wrong comparison, improper loop termination, missing exit on divergent iteration, incorrect loop variable updates.

2. Integration Testing

- **Systematic Testing**: It involves building the program structure step by step, testing for interface errors at each stage.
- **Incremental Testing**: Unlike the "big bang" approach, incremental integration tests small portions of the program gradually, which makes error detection and debugging easier.
- **Efficiency**: It ensures early detection of errors, reducing long-term costs and improving debugging efficiency.

1. **Top-Down Integration:**

- **Incremental Build**: Constructs the program structure by integrating subordinate modules into the main control module in either a **depth-first** or **breadth-first** approach.
- **Depth-First Approach**: Integrates components along a major control path first (e.g., M1, M2, M5, then M8/M6).
- **Breadth-First Approach**: Integrates all components at each level horizontally (e.g., M2, M3, M4 first).
- **Test Driver and Stubs**: The main control module is used as a test driver, with stubs replacing lower-level components initially. These stubs are gradually replaced by real modules, and tests are conducted after each integration.
- **Regression Testing**: Ensures that newly integrated components do not introduce errors into the already tested system.



Problems in Top-Down Integration:

- **Logistic Issues**: Testing upper levels early is difficult due to stubs replacing low-level modules.
- **Limited Data Flow**: Since the low-level modules are not integrated yet, data flow from the bottom to the top can be restricted.
- **Solutions**:
    - **Delay Testing**: Postpone tests until stubs are replaced.
    - **Simulate Stubs**: Develop stubs that perform limited functions to simulate the actual module.
    - **Bottom-Up Integration**: Integrate from the bottom of the hierarchy to avoid the limitation of stubs.

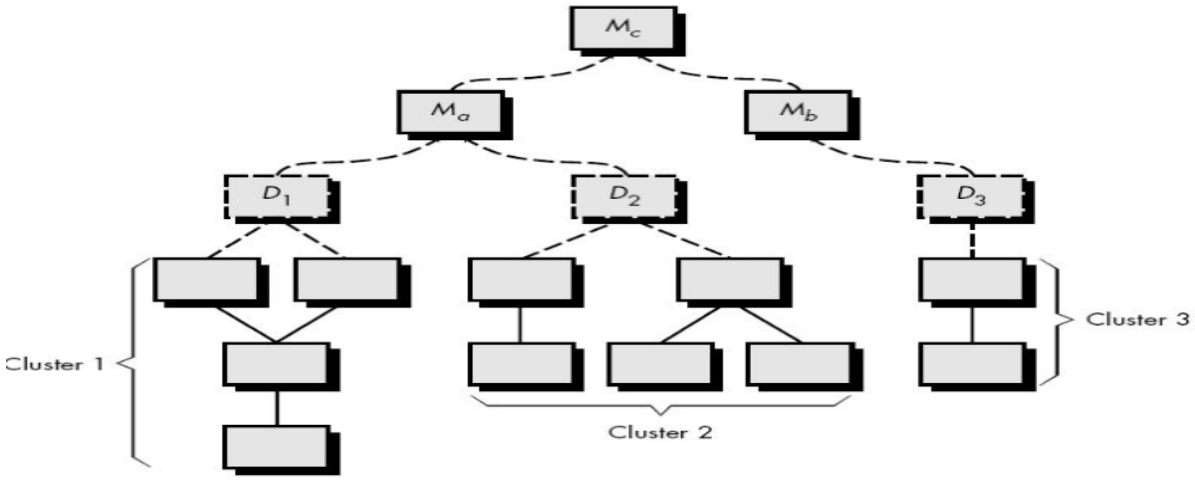
2. Bottom-Up Integration:

- **Atomic Module Testing**: Starts with testing low-level modules and gradually integrates them into higher-level modules.
- **Advantages**:
    - No stubs needed since lower-level modules are tested first.
    - Early defect detection in foundational components.

    - Thorough verification of lower-level components before higher-level modules are integrated.
- **Key Features**:
    - **Cluster Testing**: Low-level components are grouped into clusters, tested with drivers, and integrated upward in the program structure.
    - **Driver Usage**: A control program (driver) is written to handle input and output for testing.
    - **Progressive Integration**: Once clusters are tested, the drivers are removed, and the clusters are integrated into higher-level components.

Example of Bottom-Up Integration:

- **Clusters**: Components are grouped into clusters (1, 2, 3).
- **Testing**: Each cluster is tested individually with a driver.
- **Integration**: Drivers are removed, and clusters are integrated with higher-level modules (e.g., Ma with clusters 1 and 2, Mb with cluster 3).



3. Regression Testing

- **Purpose**: Regression testing re-executes previously performed tests to ensure that software changes (like bug fixes or new features) do not introduce unintended side effects, breaking existing functionality or affecting system stability.
- **Goal**: Verify that recent changes or fixes have not caused any regressions in the software.

Why is Regression Testing Necessary?:

- **Integration Changes**: During integration testing, new data flows, I/O operations, and control logic are introduced.
- **Potential Issues**: These changes can cause previously working features to stop functioning, or errors may arise that negatively impact the software's performance.

Regression Testing Steps:

1. **Identify Test Cases**: Select test cases to be re-executed based on recent changes or areas at risk.
2. **Execute Tests**: Tests can be executed manually or automatically.
3. **Compare Results**: Results are compared against expected outcomes to detect any discrepancies.
4. **Analyze Findings**: Document and investigate any failures to understand their impact and cause.

Manual vs. Automated Regression Testing:

- **Manual Testing**:
  - Re-execute a subset of test cases manually.
  - It's time-consuming but effective for exploratory or ad-hoc testing.
- **Automated Testing**:
  - Use tools that can capture and replay test cases, increasing efficiency.
  - Ideal for large test suites and frequent re-execution.
  - Helps to perform regression testing quickly and repeatedly.

Classes of Regression Test Cases:

- **Representative Sample**: A set of tests that covers the main functions of the software.
- **Additional Tests**: Focus on functions that are likely to be impacted by recent changes.
- **Changed Components**: Test cases that specifically target modified or newly added parts of the software.

Challenges in Regression Testing:

- **Test Suite Growth**: As the system becomes more complex and more features are added, the number of tests required also increases.
- **Re-execution Impracticality**: It is not always feasible to re-run all tests after every change, especially in large systems.
- **Selective and Efficient Test Suites**: There is a need to design regression tests efficiently, focusing on high-priority and likely affected areas.

Designing an Effective Regression Test Suite:

- **Focus on Error-Prone Areas**: Include tests that focus on common problem areas or parts of the system that are typically prone to errors.
- **Test Major Functions**: Ensure key functionality is thoroughly tested after every change.
- **Recent Changes**: Pay particular attention to the parts of the system that have recently been modified.
- **Efficiency**: Balance thoroughness with efficiency to save time and resources.

- **Redundancy Avoidance**: Focus on the most likely issues without duplicating tests.

Smoke Testing:

- **Definition**: Smoke testing is an integration testing approach used to quickly evaluate whether a new software build is stable enough for further testing.
- **Usage**: Commonly applied in **shrink-wrapped** software products (ready-to-use, mass-produced software).
- **Purpose**: Acts as a pacing mechanism, especially for time-critical projects, allowing teams to assess project progress frequently.

Key Features of Smoke Testing:

- **Focus**: Identify "show-stopper" errors that could block further development or testing.
- **Frequency**: Conducted **daily** after each new build.
- **Integration Approach**: Can follow **Top-Down** or **Bottom-Up** integration methods depending on project needs.

Smoke Testing Process:

1. **Build Creation**: Components like **data files**, **libraries**, **reusable modules**, and **engineered components** are integrated into a build.
2. **Error Testing**: Designed to expose **critical and major errors**.
3. **Daily Testing**: New builds are integrated with the system and tested **every day** to ensure basic functionality remains intact.

Goals of Smoke Testing:

- **Error Identification**: Detect serious issues early, preventing wasted effort on unstable builds.
- **Project Progression**: Ensure the software is stable enough to continue with deeper testing or development phases.
- **Time Management**: Save time by catching critical problems early and avoiding bigger delays later.
- **Integration Risk Minimization**: Frequent tests uncover incompatibility issues early.
- **Quality Improvement**: Helps uncover functional, architectural, and component-level defects early, leading to a higher-quality final product.
- **Simplified Error Diagnosis**: When errors are found, it's easier to trace them back to recent changes.
- **Better Progress Assessment**: Regular results provide managers and developers with a clear view of development and integration status.

Benefits of Smoke Testing:

- Fast detection of major problems.
- Saves time and resources by avoiding deep testing on unstable builds.
- Improves overall product quality.
- Makes project progress measurable and transparent.
- Simplifies debugging and error correction.

Validation Testing

- **What**: Testing if the whole software meets **user needs** and **requirements**.
- **Goal**: Confirm the software works the way customers expect.
- **Focus**: Visible features defined in the **Software Requirements Specification (SRS)**.

Includes:

1. **Validation Criteria** – Check if all requirements are fulfilled.
2. **Configuration Review** – Ensure everything is properly documented and organized.
3. **Alpha and Beta Testing** – Get real user feedback.

Configuration Review (Audit)

- Check if the system is complete, correct, and ready for maintenance.
- Done by **auditors or users**, not developers.

Acceptance Testing

- **Purpose**: Allow the customer to test and approve the software.
- **Done By**: End-users, not developers.
- **Ways**:
  - **Informal**: Quick use ("test drive").
  - **Formal**: Systematic and planned tests.

Aspect	Alpha Testing	Beta Testing
Where	At developer's site	At customer's site (real environment)
Who tests	Internal users or selected customers	Actual end-users
Purpose	Find bugs early before release	Find real-world issues and gather feedback
Environment	Controlled environment	Uncontrolled, real-world environment
Focus	Functionality, usability, and initial feedback	Reliability, performance, user experience
Developer's Role	Developers observe users and fix bugs quickly	Developers fix issues after collecting feedback

System Testing

Testing the complete integrated system to ensure everything works together properly.

Purpose:

- Confirm all components are integrated.
- Verify the system meets the required specifications.

Types of System Testing:

1. Recovery Testing

- **Checks** how well the system recovers after a failure.
- Tests **automatic recovery** like data saving, restart, and reinitialization.
- Evaluates **human recovery** time (Mean-Time-to-Repair, MTTR).

2. **Security Testing**

- **Ensures** the system protects data and prevents unauthorized access.
- Testers **simulate hackers** to find vulnerabilities.
- Common attack methods tested:
    - Stealing passwords
    - Using hacking software
    - Accessing insecure data
    - Causing system errors

3. **Stress Testing**

- **Puts extreme pressure** on the system (e.g., very high data input, heavy traffic).
- Goal: Find the system's breaking point.
- Examples:
    - Sending many interrupts per second.
    - Filling up memory completely.
    - Heavy disk usage.

4. **Performance Testing**

- **Measures** the system's speed, response time, and resource usage.
- Done **with stress testing** to evaluate overall efficiency.
- Involves tools to check CPU usage, memory, and disk activity.

Debugging

- Debugging is the process of finding and fixing errors after testing.
- Debugging is **different from testing**. It **comes after** testing.
- It matches **symptoms** (error signs) with **causes** (real problems).
- Sometimes fixing one error can **hide** or **reveal** other errors.
- Errors can be tricky:
    - Due to **human mistakes** (wrong inputs, wrong settings).
    - Due to **timing issues** (like delayed outputs).
    - Hard to reproduce, especially in **real-time or distributed systems**.

Debugging Approaches/Strategies:

Approach	Meaning
Brute Force	- Print statements, memory dumps, traces. - Easy but **slow and inefficient**. - Best for small programs or as a last resort.
Backtracking	- Start from where the error appears. - **Trace the program backward** to find the cause manually.
Cause Elimination	- List all possible causes. - Use **hypothesis and testing** to eliminate wrong causes one by one.
Correcting Error	- Fix carefully to avoid introducing new bugs. - Always check: - If similar bugs exist elsewhere. - What new bugs could arise. - How to prevent such bugs in future.

White Box Testing

- White Box Testing means testing the **internal code**, **logic**, and **structure** of a program.
- Testers **can see** and **test** how the software works from the inside.

White Box Testing Techniques

1. **Statement Coverage** - This technique is aimed at exercising all

programming statements with minimal tests.

2. **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
3. **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered

Basis Path Testing

- It is used to design test cases based on the program's **control flow**.
- First step: **Find Cyclomatic Complexity** to know how many test cases are needed.

Cyclomatic Complexity (CC)
A metric that measures the **logical complexity** of code.

Formula: **E – N + 2P**
where:

- **E** = Number of edges (connections)
- **N** = Number of nodes (steps or actions)
- **P** = Number of connected components (usually 1)

Quick Method: **CC = Number of decision points + 1**

Example:
If a flowchart has 3 decision points,
then **Cyclomatic Complexity = 3 + 1 = 4**.

Importance of Cyclomatic Complexity

- Tells **how many test cases** are needed for full coverage.
- Shows how **complex** and **risky** the program is.
- Helps find **untested paths**.
- Indicates **code readability** and **maintainability**.
- Assures that **every path is tested at least once**.

Properties of Cyclomatic Complexity

- **Maximum number of independent paths**.
- **Depends only** on the number of decisions, **not** on simple lines of code.
- Always **≥ 1**.
- **Adding/removing normal statements** (not decisions) **doesn't affect** it.

Cyclomatic Complexity	Meaning
1 – 10	·Structured and Well Written Code ·High Testability ·Less Cost and Effort
10 – 20	·Complex Code ·Medium Testability ·Medium Cost and Effort
20 – 40	·Very Complex Code ·Low Testability ·High Cost and Effort
> 40	·Highly Complex Code ·Not at all Testable ·Very High Cost and Effort

Cyclomatic Complexity Calculation Methods

1. **Method 1**:
   Cyclomatic Complexity = Total number of closed regions + 1
   - This method counts closed regions (areas where control flow loops back to an earlier point).
2. **Method 2**:
   Cyclomatic Complexity = E – N + 2
   Where:
   - **E** = Total edges (connections) in the control flow graph
   - **N** = Total nodes (steps/actions) in the control flow graph
3. **Method 3**:
   Cyclomatic Complexity = P + 1
   Where:
   - **P** = Total number of **predicate nodes** in the control flow graph
   - Predicate nodes are conditional nodes (e.g., **if**, **while**), which split the flow into multiple paths.

Graph Matrices

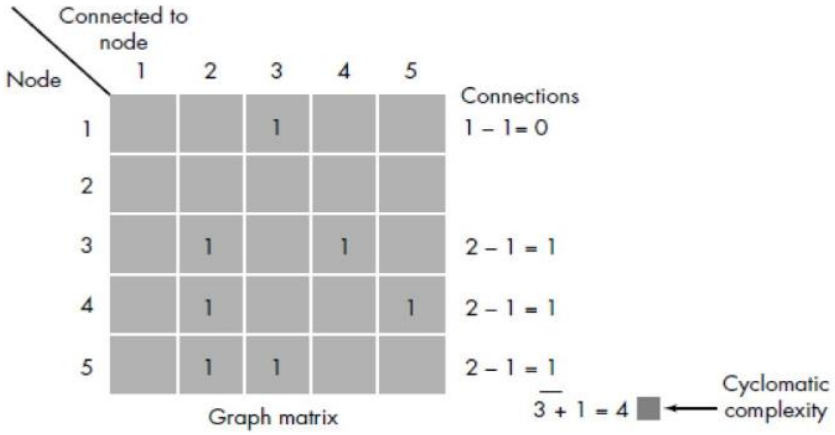
- **Control Flow Representation**:
  A matrix that helps represent the control flow of the program where each row and column represents nodes (actions).
  - **Edges** (connections between nodes) can have weights for attributes like cost or distance.
  - **Purpose**: Helps in understanding control structures and facilitates **test case generation** and **coverage** of complex code paths.

Testing Methods

1. **Condition Testing**:
   - Focuses on testing **logical conditions** in the code to ensure they behave correctly.
2. **Data Flow Testing**:

- Tests paths based on **variable definitions** and **uses**.
- Key Terms:
  - **DEF(S)**: Definitions of variable X at statement S.
  - **USE(S)**: Uses of variable X at statement S.
  - **Definition-Use (DU) chain**: Tracks when a variable is defined and later used.

Connection Matrix



Loop Testing

1. **Simple Loops**:
   - Test loop behavior with different pass conditions:
     - No passes
     - One pass
     - Multiple passes (e.g., **n-1**, **n**, **n+1** passes).
2. **Nested Loops**:
   - Start from the **innermost loop** and test it by adjusting outer loops to their minimum values.
   - Test combinations like **min+1**, **max-1**, **max** for each loop and adjust as you go outwards to the outer loops.

Black-Box Testing

Black-box testing focuses on testing the functionality of the system without knowledge of its internal workings. The key questions in Black-Box Testing include:

- **Functional Validity**: How well does the system function according to the specifications?
- **System Behavior & Performance**: How does the system behave under various conditions, and what is its performance (e.g., speed, resource usage)?
- **Input Classes for Good Test Cases**: Which classes of input (valid, invalid, boundary values) are most appropriate for testing?

- **Sensitivity to Input Values**: Does the system react in unexpected ways to certain input values?
- **Boundary Isolation**: How does the system handle inputs that are at the boundaries of valid data?
- **Data Rates and Volume**: What is the system's capacity for handling varying data volumes and data rates?
- **Combination Effects**: How does the system behave when specific combinations of data are used?

Graph-Based Methods

Graph-based methods are used to understand the relationships between objects or components in a system. These methods analyze how the various elements of software are connected.

- **Objects**: These can include data objects, traditional software components (modules), and object-oriented elements.
- **Purpose**: The goal is to gain a deeper understanding of how the software components interact and how those interactions can be tested.

Equivalence Partitioning

This is a technique used to reduce the number of test cases by grouping inputs into equivalent classes. The idea is that if one test case from a group of similar inputs passes, the others will likely pass as well.

Sample Equivalence Classes:

- **User Supplied Commands**: Commands given by the user as inputs.
- **Responses to System Prompts**: How the system reacts to user responses.
- **File Names and Computational Data**: Includes tests for valid file names, formats, and computation inputs.
- **Physical Parameters & Bounding Values**: For testing limits of acceptable values like maximum and minimum ranges.
- **Error Messages & Graphical Data**: Includes ensuring correct responses to error messages or mouse clicks.

Model-Based Testing

Model-based testing involves using a behavioral model of the system to generate test cases.

Steps:

1. **Analyze/Develop a Model**: Create or use an existing model that defines how the system should behave in response to different inputs.

2. **Specify Inputs**: Identify which inputs will cause the system to transition between different states.
3. **Expected Outputs**: Review what the expected system behavior (outputs) should be as it transitions between states.
4. **Execute Test Cases**: Run the tests and observe whether the system behaves as expected.
5. **Comparison & Action**: Compare the actual results with the expected results and make corrections if necessary.

Aspect	White Box Testing	Black Box Testing
Focus	Internal structure, logic, and code of the system	External behavior and functionality of the system
Test Basis	Based on the program's code, logic, and flow	Based on system requirements and specifications
Knowledge Required	Tester needs to know the internal workings and code structure	Tester does not need to know the internal workings of the system
Test Type	Structural testing, including path, branch, and statement coverage	Functional testing, based on user inputs and expected outputs
Testing Method	Involves inspecting code, debugging, and executing test cases based on paths	Involves providing input and observing output for correctness
Tools Used	Code analyzers, debuggers, and coverage tools	Test scripts, input-output validation tools
Test Case Design	Based on code paths, conditions, loops, and branches	Based on functional specifications and user requirements
Example	Unit testing, integration testing (code level)	System testing, acceptance testing (end-user level)
Complexity	Can be complex, requires in-depth knowledge of code	Easier to perform, does not require code knowledge
Purpose	Ensures that code behaves as expected at the code level	Ensures that the system works as expected from a user's perspective