MODULE 05 *Testing and Maintenance*

Purpose of Software Testing

- **What is Testing?**
  Testing is checking a system or its parts to see if they work as expected. It looks for differences between what we want (what should happen) and what really happens.
- **How is Testing Done?**
  Testing involves planned activities that are done in a careful and organized way.

Principles of Testing

1. **Link to Customer Needs**:
   - All tests should relate to what the customer wants.
   - Testing shows if there are errors, but it doesn't guarantee there are no other bugs.
   - The most serious issues are those that prevent the software from meeting its requirements.
2. **Exhaustive Testing is Impossible**:
   - It's not possible to test every single input and scenario.
   - Testing usually focuses on specific areas, so we need to manage our efforts wisely.
3. **Plan Tests Early**:
   - Tests should be planned before starting the actual testing.
4. **Test Early and Often**:
   - Start testing as soon as possible in the software development process.
   - Early testing helps find and fix errors sooner, making it easier to correct them.
5. **Error Clusters**:
   - Errors aren't spread evenly; if one error is found, there might be more nearby.
   - Testing needs to be flexible to adjust to this.
6. **Decreasing Effectiveness**:
   - Over time, the effectiveness of tests can decrease.
   - If we keep using the same test cases, they might not find new errors.
   - To avoid this, we should change and update our test cases regularly.
7. **Context Matters**:
   - No two systems are the same, so testing must be tailored to each system.
8. **No Errors Doesn't Mean Usable**:

- Finding and fixing errors doesn’t always mean the system will meet users' needs.
- Involving users early and using prototypes can help avoid problems later.

9. **Pareto Principle**:
    - In testing, often a small number of issues cause most of the problems.
10. **Start Small, Go Big**:
    - Begin testing with smaller parts of the software and then move to the larger system.
11. **Independent Testing**:
    - For best results, testing should be done by someone not involved in the development process.

Goals of Testing

The testing process has two main goals:

1. **Show Software Meets Requirements**:
    - The first goal is to prove to both the developer and the customer that the software works as it should.
    - This is called **validation testing**, where we check if the system performs correctly using test cases that match how it’s expected to be used.
2. **Find Errors and Issues**:
    - The second goal is to identify situations where the software behaves incorrectly or doesn’t meet its specifications. This is known as **verification**.
    - This leads to **defect testing**, where test cases are specifically designed to find bugs.
    - It includes activities that ensure the software correctly performs specific functions or algorithms.

Testing Concepts

- **Test Component**:
  A part of the system that can be tested on its own.
- **Fault (Bug or Defect)**:
  A mistake in design or code that can lead to unexpected behavior in the system.
- **Erroneous State**:
  A sign that there is a fault when the system is running.
- **Failure**:
  When the software behaves differently than expected. A failure happens because of one or more erroneous states.

Requirements of Testing

1. **Testability**:
    - Testability means how easily we can test a software program.
    - There are ways to measure how testable a program is.
2. **Operability**:
    - The better the software works, the easier it is to test.
    - Fewer bugs lead to more efficient testing since bugs make testing harder.
3. **Observability**:
    - You need to see clear outputs for each input.
    - The state of the system and its variables should be visible while it's running.
    - You should be able to look back at past states (like using logs).
    - All factors that affect the output should be visible, making it easy to identify incorrect outputs.
    - Internal errors should be detected and reported automatically.
    - The source code should be accessible for review.
4. **Controllability**:
    - Better control over the software allows for more automated and optimized testing.
    - You should be able to generate all possible outputs by changing inputs.
    - All parts of the code should be executable through various inputs.
    - Test engineers should be able to directly control the software and hardware states.
    - Input and output formats should be clear and consistent.
    - Tests should be easy to specify, automate, and repeat.
5. **Decomposability**:
    - Breaking down testing into smaller parts helps isolate problems more quickly.
    - The software should be made up of independent modules that can be tested on their own.
6. **Simplicity**:
    - The less complex the software, the faster it can be tested.
    - Simplicity in features, architecture, and code helps in easier testing and maintenance.
7. **Stability**:
    - Fewer changes to the software lead to fewer disruptions in testing.

    - Changes should happen infrequently and be controlled so they don't invalidate existing tests.
    - The software should recover well from failures.

8. **Understandability**:
    - The more we understand the software, the better we can test it.
    - The design, dependencies, and changes should be well understood and communicated.
    - Technical documentation should be easy to access, organized, detailed, and accurate.

Test Case

- **What is a Test Case?**
  A test case is a set of conditions used to check if a system meets its requirements and works correctly. Creating test cases can also help identify problems in the requirements or design.
- **Attributes of a Test Case**:
    1. **Name**:
       This helps identify the test case. For example, if you're testing a function called Deposit(), you might name the test case Test_Deposit.
    2. **Location**:
       This shows where to find the test case, like a path name or URL to the test program and its inputs.
    3. **Input (Data)**:
       This includes the specific data or commands that will be used during the test. It can refer to test data or links to where the data can be found.
    4. **Expected Behavior**:
       This is what you expect to happen during the test, described by the oracle attribute.
    5. **Log**:
       This is a record of what actually happened during the test, with timestamps to compare the observed behavior to the expected behavior.

Strategies for Software Testing

1. **Start Small, Then Expand**:
   Testing begins at the component level (individual parts) and then moves outward to test how these components work together in the whole system.
2. **Use Different Techniques**:
   Different testing methods are suitable at various stages of development.

3. **Who Tests?**:
   Testing is done by the software developers and, for larger projects, by an independent testing team.
4. **Testing vs. Debugging**:
   Testing and debugging are separate activities, but debugging should be part of any testing strategy.
   - A good strategy includes both low-level tests (testing individual components) and high-level tests (testing the system as a whole).
   - The strategy should guide testers and provide milestones for project managers to track progress.

Strategic issues in testing and maintenance

1. Specify Product Requirements in a Quantifiable Manner

- **Overview**: Clearly defining product requirements before testing is crucial. This means requirements should be measurable (e.g., performance metrics, user capacity).
- **Objective**: The primary goal of testing is to identify errors, but a comprehensive strategy also evaluates other quality factors like usability and performance.
- **Benefit**: Measurable requirements lead to clear, unambiguous results, making it easier to determine if the software meets its goals.

2. State Testing Objectives Explicitly

- **Overview**: Testing objectives need to be clearly articulated and quantifiable within the test plan.
- **Examples of Objectives**:
  - **Test Effectiveness**: Evaluates how well tests identify defects.
  - **Test Coverage**: Assesses the extent to which the software’s code and requirements are tested.
  - **Cost to Find and Fix Defects**: Analyzes the resources required to discover and resolve issues.
  - **Frequency of Occurrence**: Tracks how often specific defects are found.
  - **Test Work-Hours**: Measures the time invested in testing activities.

3. Understand the Users of the Software

- **Overview**: Developing user profiles helps tailor testing efforts to actual user interactions with the software.
- **Use Cases**: Describing how different users will interact with the software can pinpoint critical areas for testing.

- **Benefit**: This focus reduces overall testing effort while ensuring that the software effectively meets user needs.

4. Develop a Testing Plan Emphasizing Rapid Cycle Testing

- **Overview**: This approach involves a mindset that prioritizes quick, efficient testing cycles.
- **Objective**: Aim for fast and cost-effective testing that yields high-quality results.
- **Feedback Loop**: Rapid testing allows for immediate feedback, enabling adjustments in both quality control and testing strategies.

5. Build "Robust" Software

- **Overview**: Software should be designed with self-diagnostic capabilities to identify and manage certain errors autonomously.
- **Automation**: The design should support automated and regression testing, facilitating easier maintenance and updates.
- **Benefit**: This leads to more reliable software that requires less manual intervention in error detection.

6. Use Effective Technical Reviews as a Filter Prior to Testing

- **Overview**: Conducting technical reviews can uncover errors early in the development process.
- **Advantage**: These reviews can be as effective as formal testing and help reduce the amount of subsequent testing needed, thus saving time and resources.

7. Develop a Continuous Improvement Approach for the Testing Process

- **Overview**: Testing strategies should be continuously measured and improved.
- **Metrics**: Data collected during testing should inform a statistical process control approach, enabling ongoing refinement of testing methodologies.
- **Goal**: This leads to a more effective and efficient testing process over time.

Summary

Addressing these strategic issues ensures a more robust and effective software testing process. By focusing on quantifiable requirements, clear objectives, user understanding, rapid testing cycles, robust software design, effective reviews, and continuous improvement, you can significantly enhance the quality and reliability of the software developed.

Verification and Validation

1. **Verification**:
   - **Definition**: Ensures the software correctly implements specific functions or algorithms.

    - **Focus**: Checking if the product is built right (i.e., meets design specifications).
2. **Validation**:
    - **Definition**: Ensures that the software meets customer requirements and expectations.
    - **Focus**: Checking if the right product is built (i.e., fulfills user needs).

SQA Activities

Software Quality Assurance includes various activities to ensure quality:

- **Formal Technical Reviews**: Systematic examination of software artifacts.
- **Quality and Configuration Audits**: Assessing adherence to quality standards and configuration management.
- **Performance Monitoring**: Tracking software performance against defined criteria.
- **Simulation**: Running scenarios to test software behavior under various conditions.
- **Feasibility Study**: Evaluating if the project is viable within constraints (time, budget).
- **Documentation Review**: Checking the completeness and accuracy of documentation.
- **Database Review**: Assessing database structure and data integrity.
- **Algorithm Analysis**: Evaluating algorithms for correctness and efficiency.
- **Development Testing**: Testing during the software development process.
- **Qualification Testing**: Verifying that the software meets its specifications.
- **Installation Testing**: Ensuring the software installs correctly in its intended environment.

Summary

- **Testing** plays a key role in quality assurance and error detection by applying principles that enhance software quality.

Formal Technical Reviews (FTR)

- **Definition**: A Formal Technical Review (FTR) is a structured quality control activity involving software engineers and other stakeholders. It's designed to assess various aspects of software representations (like code, design documents, etc.).

Objectives of FTR

1. **Error Detection**: Identify errors in function, logic, or implementation across any software representation.
2. **Requirements Verification**: Ensure the software meets its specified requirements.
3. **Standards Compliance**: Confirm that the software adheres to predefined standards.

4. **Uniform Development**: Promote consistency in the software development process.
5. **Manageability**: Make projects easier to manage through regular oversight.

Training Opportunity

- FTRs also serve as a training ground for junior engineers, allowing them to observe different methodologies in software analysis, design, and implementation.

Types of Reviews

- FTR is a category of reviews that includes **walkthroughs** (informal presentations of the software) and **inspections** (more formal evaluations).

Review Meeting Structure

- **Participants**: Typically involves 3 to 5 people, including the review leader, reviewers, and the producer of the software. One reviewer acts as the recorder.
- **Preparation**: Each participant should prepare in advance for no more than two hours.
- **Duration**: The meeting itself should last less than two hours.

Meeting Process

- The producer presents the software and walks the team through it.
- At the end, attendees decide whether to:
    1. Accept the product without modifications.
    2. Reject the product due to significant errors.
    3. Accept with minor revisions.

Reporting and Record Keeping

- During the review, the recorder documents all raised issues and produces a review issues list, including:
    - What was reviewed
    - Who reviewed it
    - Findings and conclusions

Review Guidelines

Here’s a minimum set of guidelines to ensure effective FTRs:

1. **Review the Product, Not the Producer**: Focus on the work, not the individual who produced it.
2. **Set and Maintain an Agenda**: Keep the meeting organized and on schedule.
3. **Limit Debate**: Avoid prolonged discussions about issues raised; instead, focus on identifying problems.
4. **Identify Problems**: Point out issues without attempting to solve them during the meeting.

5. **Take Written Notes**: Use a wall board for visibility of notes and priorities.
6. **Limit Participants and Require Preparation**: Keep the team small and ensure everyone is prepared.
7. **Develop Checklists**: Create checklists for common review items to guide discussions.
8. **Allocate Resources**: Schedule FTRs as essential tasks in the software development process.
9. **Training for Reviewers**: Ensure all participants receive proper training to contribute effectively.

Summary

FTRs are vital for maintaining software quality and fostering team collaboration. By adhering to structured processes and guidelines, they help uncover issues early in the development cycle and facilitate effective communication among team members.

Levels of Testing (Unit Testing)

- **Definition**: Unit Testing is a level of software testing that focuses on validating individual units or components of the software. A "unit" typically refers to the smallest testable part of the application, like a function or method.

Key Characteristics of Unit Testing

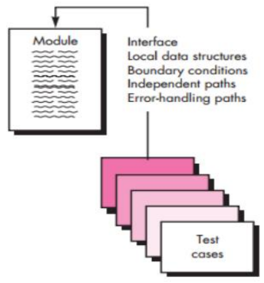
1. **Functional Correctness**: The primary goal is to ensure that each standalone module functions correctly according to its specifications.
2. **Control Path Testing**: Important execution paths within the module are tested to identify potential errors or unexpected behavior.
3. **Limited Scope**: The tests are designed to cover a constrained scope, which means they may not reveal all errors present in the overall system. The focus is on the unit's functionality rather than interactions with other units.
4. **Internal Logic Focus**: Unit Testing examines the internal logic and data structures used within the module. This means it looks at how data is processed and managed internally.
5. **Parallel Testing**: Multiple components can be tested simultaneously, which allows for efficient use of time and resources during the development process.

Why Unit Testing is Important

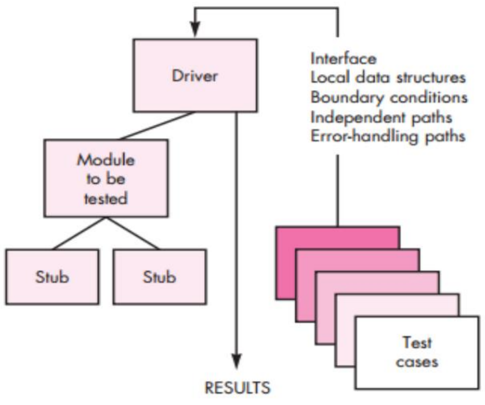
- **Early Error Detection**: By testing individual components early in the development cycle, developers can catch and fix bugs before they propagate to higher levels of testing, which can be more costly and time-consuming.
- **Facilitates Change**: Unit tests serve as a safety net that allows developers to refactor or modify code with confidence, knowing that any introduced errors can be quickly identified through failing tests.

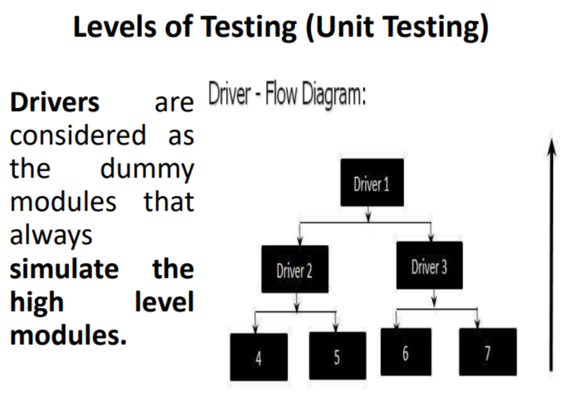
- **Documentation**: Unit tests can act as documentation for the code, providing clear examples of how each unit is intended to behave.

- Unit Test



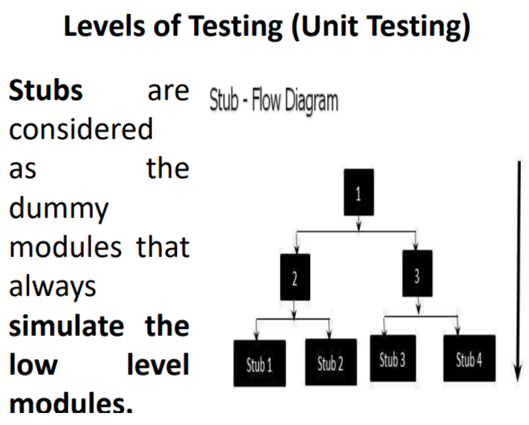
- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- All independent paths through control structure exercised to ensure all statements executed.
- Boundary conditions are tested to limit or restrict processing.
- All error-handling paths are tested
- Data flow across a component interface is tested before any other testing.
- Test cases should be designed to uncover errors due to improper computation , comparison and data flow.
- Boundary testing is one of the most important unit testing tasks. Levels of Testing (Unit Testing)
- Errors often occur when the maximum or minimum allowable value is encountered (boundary testing)
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors





Levels of Testing (Unit Testing)

Drivers are considered as the dummy modules that always **simulate the high level modules.**



Levels of Testing (Unit Testing)

Stubs are considered as the dummy modules that always **simulate the low level modules.**

- Drivers and stubs represent testing “overhead.”
- Both are software that must be written but that is not delivered with the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low.
- Unit testing is simplified when a component with high cohesion is designed

Levels of Testing (Integration Testing)

Integration Testing is crucial for several reasons:

- **Data Loss Across Interfaces**: Issues may arise when components communicate, leading to potential data loss.

- **Adverse Effects**: One component can adversely affect another, causing unexpected behaviors.

- **Combining Sub Functions**: When smaller functions are combined, they might not work together as intended.

- **Global Data Structures**: Shared data structures can introduce problems, especially when accessed concurrently.

Overview of Integration Testing

- **Systematic Technique**: Integration Testing systematically constructs the software architecture while testing for errors associated with interfacing between components.

- **Avoiding Big Bang Integration**: A "big bang" approach, where all components are integrated at once, is risky and can lead to complex issues, such as endless loops.

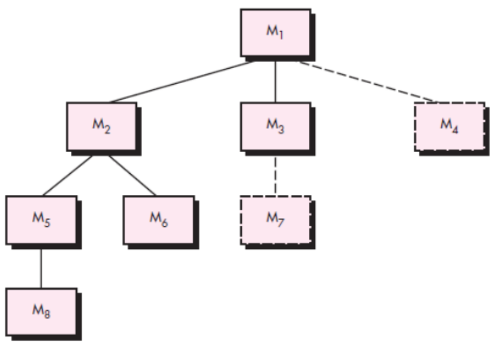
Incremental Integration

- **Definition**: The program is constructed and tested in small increments, making it easier to isolate and fix errors.

- **Benefits**: This method ensures more complete testing of interfaces.

Incremental Integration Strategies

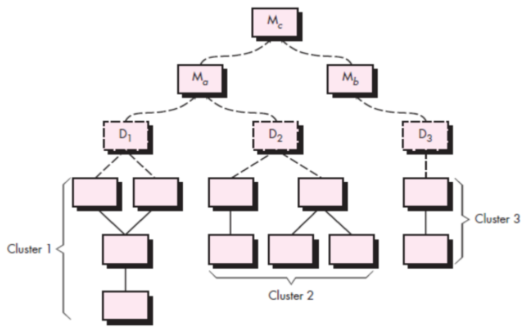
1. **Top-Down Integration**:
    - **Approach**: Top-level units are tested first, followed by lower-level units.
    - **Integration Steps**:
        1. The main control module acts as a test driver, while stubs substitute for all components directly subordinate to it.
        2. Depending on whether a depth-first or breadth-first approach is chosen, subordinate stubs are replaced one at a time with actual components.
        3. Tests are conducted as each component is integrated.
        4. After testing, another stub is replaced with the real component.
        5. Regression testing may occur to ensure no new errors were introduced.
    - **Advantages**: This strategy verifies major control or decision points early in the testing process.
    - **Challenges**: The top-down approach can struggle when low-level processing is needed to test upper levels adequately.
    - Top – down integration.

2. **Bottom-Up Integration**:
    - **Approach**: Bottom-level units are tested first, with upper-level units integrated step by step.
    - **Elimination of Stubs**: Because components are integrated from the bottom up, there's no need for stubs.
    - **Program Completeness**: The entire program only exists once the last module is integrated.
    - **Integration Steps**:
        1. Low-level components are combined into clusters (sometimes called builds) that perform specific software sub-functions.

2. A driver (a control program for testing) is written to manage test case inputs and outputs.
3. The cluster is tested.
4. Drivers are removed as clusters are combined moving upward in the program structure.

Bottom Up Integration



Levels of Testing (Regression Testing)

- **Definition**: Regression testing is a type of software testing aimed at ensuring that changes (like enhancements or defect fixes) have not adversely affected the existing functionality of the software.

Importance of Regression Testing

- **Changes and Effects**: When a new module is added or changes are made during integration testing, the software's behavior may change. This could introduce new data flow paths, new input/output methods, and new control logic.
- **Potential Problems**: Changes made to the software may inadvertently affect functions that previously worked correctly. This makes it essential to verify that existing functionalities remain intact after modifications.

Purpose of Regression Testing

- **Re-execution of Tests**: Regression testing involves re-running a subset of previously conducted tests to check for any unintended side effects resulting from the recent changes.
- **Error Discovery and Correction**: When errors are found and corrected, it may alter some aspects of the software configuration, necessitating further testing to ensure stability.
- **Ensures Stability**: The primary goal of regression testing is to ensure that new changes do not introduce additional errors into the system.

Execution of Regression Testing

- **Manual or Automated**: Regression testing can be conducted manually or through automated tools that capture playback scenarios, which helps in efficient testing.

Effective Regression Testing

To create an effective regression test suite, it should include three different classes of test cases:

1. **Representative Sample**: A set of tests that exercises all software functions to ensure overall functionality.
2. **Focused Tests**: Additional tests specifically targeting the software functions that are likely to be affected by recent changes. This ensures that areas most impacted by modifications are thoroughly tested.
3. **Component-Specific Tests**: Tests focusing on the specific components that have undergone changes. This is critical to verify that the modifications did not introduce new issues.

Summary

Regression testing is a vital process in software development that helps maintain the integrity of existing functionality after changes are made. By systematically re-testing previously executed tests, developers can ensure that enhancements or fixes do not lead to new errors, thereby enhancing software reliability.

Levels of Testing (Acceptance Testing)

- **Definition**: Acceptance testing is a testing technique aimed at determining whether the software system meets its requirement specifications.

Purpose of Acceptance Testing

- **Evaluate Business Requirements**: The primary purpose is to assess if the system fulfills the business requirements and is ready for delivery to end users.
- **Testing Method**: Typically, the black box testing method is used, focusing on the system's outputs based on given inputs without concern for its internal workings.

Types of Acceptance Testing

1. **Benchmark Testing**:
    - **Description**: In benchmark testing, the client prepares a set of test cases that represent typical operating conditions for the system. This helps ensure that the system can handle real-world scenarios effectively.
2. **Competitor Testing**:
    - **Description**: This involves testing the new system against an existing system or competitor products. It helps assess the new system's performance relative to others on the market.
3. **Shadow Testing**:
    - **Description**: A form of comparison testing where the new and legacy systems are run in parallel. Their outputs are compared to ensure the new system performs as expected alongside the existing one.

Post-Acceptance Testing Process

- **Client Feedback**: After acceptance testing, the client communicates to the project manager any unmet requirements or issues identified during testing.
- **Dialogue Opportunity**: Acceptance testing provides a platform for dialogue between developers and clients, fostering collaboration and understanding regarding the system's performance.
- **Iteration Basis**: If any requirements need to change, these changes form the basis for another iteration of the software life-cycle process.
- **Acceptance**: If the customer is satisfied with the system's performance and functionality, the system is officially accepted.

Levels of Testing (White Box Testing)

- **Definition**: White Box Testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing, or Structural Testing) is a software testing method in which the internal structure, design, and implementation of the item being tested are known to the tester.

Objectives of White Box Testing

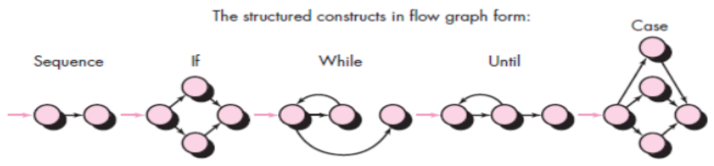
Using white-box testing methods, you can derive test cases that:

1. **All Independent Paths**: Ensure all independent paths within a module have been exercised.
2. **Logical Decisions**: Exercise all logical decisions on their true and false sides.
3. **Loop Execution**: Execute all loops at their boundaries and within their operational bounds.
4. **Internal Data Structures**: Exercise internal data structures to ensure their validity.

Basis Path Testing

- **Technique**: Basis path testing is a white-box testing technique that focuses on complexities, execution paths, data flow, independent paths, and procedural design.
- **Guarantee**: Test cases derived from this method guarantee that every statement in the program is executed at least once.

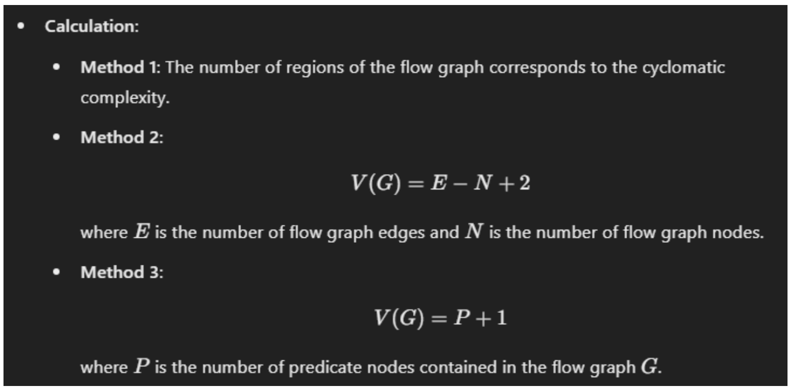
Flow Graph Notation

- **Graph Structure**:
    - 

    - Each circle, called a flow graph node, represents one or more procedural statements.
    - Edges or links represent the flow of control and are analogous to flowchart arrows.
- **Regions**: Areas bounded by edges and nodes are referred to as regions.

- **Predicate Nodes**: Each node that contains a condition is called a predicate node.
- **Independent Paths**: An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Cyclomatic Complexity

- **Definition**: Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
- **Significance**: It defines the number of independent paths, which can be further used in the development of test cases.
- 

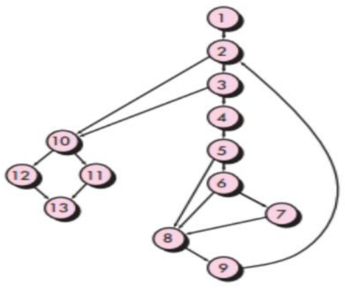
  - **Calculation:**
    - **Method 1:** The number of regions of the flow graph corresponds to the cyclomatic complexity.
    - **Method 2:**

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

      where E is the number of flow graph edges and N is the number of flow graph nodes.
    - **Method 3:**

      $$V(G) = P + 1$$

      where P is the number of predicate nodes contained in the flow graph G.

- 

White Box Test Design Procedure

1. **Draw Flow Graph**: Using the design or code as a foundation, create a flow graph.
   - **Diagram**: Flow graph illustrating the process.
2. **Determine Cyclomatic Complexity**: Calculate the cyclomatic complexity of the resultant flow graph.
3. **Identify Independent Paths**: Determine a set of linearly independent paths. The value of V(G) provides an upper bound on the number of independent paths.
4. **Prepare Test Cases**: Develop test cases that will force execution of each path in the set.

  - Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.

5. **Execute and Compare**: Each test case is executed and compared to expected results, ensuring that all statements in the program have been executed at least once.

   Additional Tools

- **Graph Matrix**: A data structure known as a graph matrix can be quite useful for developing a software tool that assists in basis path testing.

Black-Box Testing

- **Definition**: Black-box testing, also known as behavioral testing, evaluates the software based on its functionality rather than its internal code structure. Testers do not need knowledge of the internal workings; they only focus on input-output behavior.

Key Features:

- **Focus on Functional Requirements**: The primary goal is to ensure that the software meets its specified functional requirements. This means verifying that the system does what it's supposed to do from a user's perspective.

Categories of Errors Found:

Black-box testing aims to identify several types of errors, including:

1. **Incorrect or Missing Functions**: Tests ensure that all required functions are implemented correctly and are available to users.
2. **Interface Errors**: Checks whether different software components interact properly, including user interfaces, APIs, and external systems.
3. **Errors in Data Structures or External Database Access**: Verifies that data is correctly handled, ensuring proper storage, retrieval, and manipulation when interacting with databases.
4. **Behavior or Performance Errors**: Assesses the software's performance under various conditions, checking for issues like slow response times or system crashes.
5. **Initialization and Termination Errors**: Tests whether the software starts and stops correctly, ensuring no errors occur during initialization or shutdown.

Relationship to White-Box Testing:

- **Complementary Approach**: Black-box testing is not a replacement for white-box testing (which involves testing internal structures and code). Instead, it complements white-box techniques, revealing different types of errors. Together, they provide a more comprehensive testing strategy.

Criteria for Effective Testing:

- **Identifying Classes of Errors**: Focus on recognizing different categories of potential errors to ensure thorough coverage.

- **Designing Additional Test Cases**: It’s essential to create various test cases to adequately cover functional requirements and edge cases, ensuring that the software is tested thoroughly and effectively.

Summary:

Black-box testing is a crucial part of the software testing lifecycle, emphasizing functional correctness without delving into internal code. By focusing on user requirements and system behavior, it helps identify errors that might not be evident through other testing methods.

Graph-Based Testing Methods

Graph-based testing methods are approaches that model software using graphs to represent objects and the relationships between them. This method helps visualize and analyze how different components of the software interact.

Key Concepts:

- **Modeling Objects and Relationships**: The first step is to identify the important objects within the software and the relationships that connect them. This visualization helps in understanding the structure and behavior of the system.
- **Creating a Graph**: A graph is created to represent these objects and their relationships. Each node in the graph represents an object, while edges represent the relationships between them.
- **Defining Test Cases**: Test cases are designed to ensure that all objects are functioning correctly and maintaining the expected relationships. This involves covering the entire graph to uncover potential errors.

Types of Graph-Based Testing Methods:

Here are the specific graph-based testing methods mentioned in your notes:

1. **Transaction Flow Modeling**:
    - **Description**: Focuses on the flow of transactions within the system.
    - **Example**: In an airline reservation system, the flow might include steps like searching for flights, selecting a flight, and validating payment. Data flow diagrams can help create these transaction flow graphs.
2. **Finite State Modeling**:
    - **Description**: Models the various states that the software can be in, particularly from a user's perspective.
    - **Example**: When a user checks order information, the system might transition from checking availability to verifying customer billing information. State transition diagrams are used to visualize these state changes.
3. **Data Flow Modeling**:

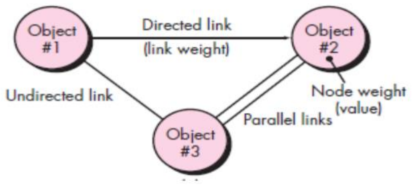
- **Description**: Focuses on how data is transformed as it moves through the system. This method looks at how one data object is converted into another.
- **Purpose**: Helps identify any issues that arise during data transformations, ensuring that data flows correctly through the system.

4. **Timing Modeling**:
   - **Description**: Examines the timing aspects of interactions between objects.
   - **Focus**: This method specifies the required execution times for processes, ensuring that operations occur in a timely manner and adhere to performance requirements.

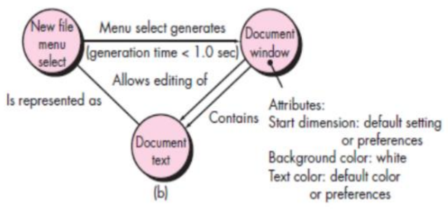
Summary:

Graph-based testing methods provide a structured approach to software testing by using visual representations of objects and their relationships. By leveraging different modeling techniques, testers can create comprehensive test cases that explore various aspects of software behavior, helping to identify errors in functionality, state transitions, data handling, and timing. This approach enhances the thoroughness of testing and ensures a more robust software product.

- Graph notation



- Example



Equivalence Partitioning

Equivalence partitioning is a testing technique that divides input (or output) data into distinct partitions, or equivalence classes, from which test cases can be derived. The goal is to identify representative test cases that can cover a range of possible inputs, minimizing the total number of tests while maximizing coverage.

Key Concepts:

- **Partitions of Data**: The data is divided into groups (partitions) that can be treated as equivalent for the purpose of testing. Each partition should ideally lead to similar behavior in the software.
- **Deriving Partitions**: Equivalence partitions are typically derived from the requirements specification, focusing on input conditions that will be tested.

Benefits:

- **Error Detection**: This technique helps uncover classes of errors by ensuring that test cases cover both valid and invalid inputs.

- **Efficiency**: By reducing the number of test cases needed while still covering all scenarios, equivalence partitioning makes the testing process more efficient.

Equivalence Classes:

An equivalence class is a set of inputs that are treated similarly by the software. The input conditions can be categorized as follows:

1. **Numeric Value Range**: For a range (e.g., 1 to 100):
   - **Valid Class**: Any number between 1 and 100 (e.g., 50).
   - **Invalid Classes**: Numbers less than 1 (e.g., 0) and numbers greater than 100 (e.g., 101).
2. **Specific Value**: For a condition requiring a specific value (e.g., 5):
   - **Valid Class**: The number 5.
   - **Invalid Classes**: Any number other than 5 (e.g., 4 and 6).
3. **Member of a Set**: For conditions that specify a set (e.g., {A, B, C}):
   - **Valid Class**: Any member of the set (e.g., A).
   - **Invalid Class**: Any value not in the set (e.g., D).
4. **Boolean Condition**: For a true/false condition:
   - **Valid Class**: True.
   - **Invalid Class**: False.

Guidelines for Creating Equivalence Classes:

1. **Range Inputs**: For a range, define one valid and two invalid classes.
2. **Specific Value**: For a specific value, define one valid and two invalid classes.
3. **Set Membership**: For set membership, define one valid and one invalid class.
4. **Boolean Inputs**: For Boolean conditions, define one valid and one invalid class.

Test Case Selection:

- The goal is to create test cases that cover the maximum number of attributes of an equivalence class simultaneously. This means selecting representative values that effectively validate the behavior of the software.

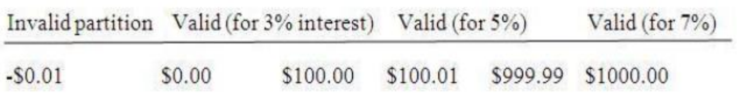
Example:

Consider a savings account with different interest rates based on balance:

- If the balance must be between $1,000 and $10,000 to receive a certain interest rate:
  - Valid Class: A balance of $5,000.
  - Invalid Classes: A balance of $500 (too low) and $15,000 (too high).

Summary:

Equivalence partitioning is a powerful technique in software testing that helps to ensure comprehensive coverage of input conditions while minimizing redundancy. By strategically selecting test cases from defined equivalence classes, testers can efficiently identify errors and verify that the software behaves as expected across a variety of scenarios.



Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
-$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

Boundary Value Analysis (BVA)

Boundary Value Analysis is based on the observation that most errors occur at the boundaries of input domains rather than in the center. Therefore, this technique focuses on selecting test cases that specifically exercise these boundary values.

Key Concepts:

- **Error-Prone Boundaries**: Many software errors are likely to occur at the edges of input ranges, such as minimum and maximum values, rather than within the middle of those ranges.
- **Selection of Test Cases**: BVA leads to the selection of test cases that target these boundary conditions, ensuring thorough testing of critical points in the input domain.

Application Beyond Inputs:

- BVA is not limited to input conditions; it can also be applied to output domains. For example, when examining the relationship between temperature and pressure in a system, boundary values for both metrics could be tested.

Examples:

1. **Temperature vs. Pressure Table**: When testing how temperature affects pressure, BVA would involve testing values at and near the critical temperature points where changes in pressure might occur.
2. **Internal Data Structures**: If a program has internal data structures that require specific boundaries (like an array with a defined size), BVA would involve testing the limits of that size.

Guidelines for Boundary Value Analysis:

1. **Range Bounded by Values a and b**:
    - Test cases should be designed for:
        - The exact lower boundary value a
        - The exact upper boundary value b
        - A value just below the lower boundary (i.e., a−1)
        - A value just above the upper boundary (i.e., b+1)

This creates a set of test cases that effectively cover all critical boundary conditions.

2. **Number of Values**:
    - When an input condition specifies a number of values (e.g., a system allows for 1 to 10 items), test cases should be designed for:
        - The minimum value (1)
        - The maximum value (10)
        - Values just below the minimum (0) and just above the maximum (11)

Summary:

Boundary Value Analysis is a targeted approach that focuses on the extremes of input and output ranges to uncover potential errors. By systematically testing values at the boundaries, as well as just inside and outside those boundaries, BVA ensures that critical edge cases are validated, significantly enhancing the robustness of software testing. This method helps in identifying defects that might not be found by merely testing the middle of the input range.

Overview of OOA and OOD Models

- **Construction of Object-Oriented Software**: The development process starts with creating analysis and design models based on requirements. These models help define the system's structure and behavior before implementation.
- **Importance of Review**: Reviewing OOA and OOD models is crucial because it can help avoid common pitfalls that might arise during analysis and design stages.

Problems Avoided Through Early Review

During Analysis:

1. **Avoiding Special Subclasses**: Prevents the unnecessary creation of subclasses to handle invalid attributes, which can complicate the class hierarchy.
2. **Misinterpretation of Class Definitions**: Helps clarify class definitions, ensuring that relationships between classes are accurate and relevant.
3. **Characterization of System Behavior**: Ensures that system behavior accurately reflects intended functionality without accommodating extraneous attributes.

During Design:

1. **Proper Class Allocation**: Helps ensure that classes are correctly assigned to subsystems and tasks during the design phase.
2. **Avoiding Unnecessary Design Work**: Prevents extra work related to creating procedural designs for operations that handle irrelevant attributes.
3. **Accurate Messaging Model**: Ensures that the messaging model, which defines how classes communicate, is correct.

Importance of Rigorous Review

- Both OOA and OOD models provide significant insights into the system's structure and behavior, making rigorous reviews essential before coding begins.

Correctness of OOA and OOD Models

- **Syntactic Correctness**: Ensures proper use of symbols and modeling conventions.
- **Semantic Correctness**: The model should accurately reflect real-world entities and relationships.

Consistency of Object-Oriented Models

- **Evaluating Consistency**: Consistency is assessed by examining the relationships among entities in the model. The connections between classes should be logical and coherent.
- **Class-Responsibility-Collaboration (CRC) Model**: This model is used to measure consistency by detailing each class's responsibilities and its interactions with other classes.

Steps to Evaluate the Class Model

1. **Review the CRC Model and Object-Relationship Model**: Revisit these models to ensure that they align with the requirements.
2. **Inspect CRC Index Cards**: Verify that responsibilities delegated to collaborators are defined correctly within their descriptions.
3. **Invert Connections**: Ensure that each collaborator receiving requests for services is appropriately connected to the source making the request.
4. **Validate Classes and Responsibilities**: Determine if the classes are valid and if responsibilities are appropriately distributed among them.
5. **Combine Responsibilities When Necessary**: Identify if widely requested responsibilities can be consolidated into a single responsibility for efficiency.

Summary

The review process for OOA and OOD models is critical in ensuring that the software is built on a solid foundation of accurate, consistent, and meaningful representations of the system. By addressing potential issues early in the development cycle and adhering to established guidelines for correctness and consistency, teams can avoid costly errors and design flaws, leading to a more robust software product.

Object-Oriented Testing Strategies Overview

Object-oriented (OO) testing strategies differ from classical software testing due to the nature of OO systems, which are based on classes and objects. These strategies typically move from "testing in the small" (unit testing) to "testing in the large" (integration and validation testing).

1. Unit Testing in the OO Context

- **Definition**: In OO testing, the smallest testable unit is the class rather than individual functions or methods.

- **Classes and Objects**: Each class has attributes and operations. Since classes often inherit operations from superclasses, it's important to test these inherited operations within the context of the subclass.
- **Contextual Testing**:
    - Testing an operation (e.g., X()) solely in isolation is ineffective. The operation might behave differently in subclasses due to overridden methods or additional attributes.
    - Therefore, unit testing must consider both the encapsulated operations and the state behavior of the class, ensuring that inherited operations are tested in all relevant contexts.

2. Integration Testing in the OO Context

Integration testing focuses on combining and testing multiple classes to ensure they work together correctly. There are two main strategies for this:

2.1 Thread-Based Testing

- **Description**: This strategy integrates classes required to respond to a specific input or event in the system.
- **Process**: Each integration thread (a group of classes) is tested individually to check for side effects and ensure that each component functions as expected when combined.

2.2 Use-Based Testing

- **Description**: This strategy begins by testing independent classes first and then moves to dependent classes that utilize those independent classes.
- **Process**: The testing continues layer by layer through dependent classes until the entire system is constructed. This approach avoids the use of drivers and stubs, which can complicate testing.

3. Validation Testing in an OO Context

- **Focus**: Validation testing in OO systems emphasizes user-visible actions and recognizable outputs.
- **Use Cases**: Testers should base their validation tests on use cases derived from the requirement model. These use cases provide scenarios that are likely to reveal errors in user interaction requirements.
- **Goal**: The aim is to ensure that the system meets user expectations and functions correctly from the user's perspective.

Software Rejuvenation

This is about keeping software healthy and up-to-date. It includes:

Re-documentation

- **What**: Updating or creating new documentation for the software.

- **Why**: Helps people understand how the software works.
- **Output**: Charts and tables that show how parts of the software interact.

Restructuring

- **What**: Changing the code's organization without altering how it works.
- **Why**: Makes the code cleaner and easier to maintain.
- **Example**: Breaking down big functions into smaller, simpler ones.

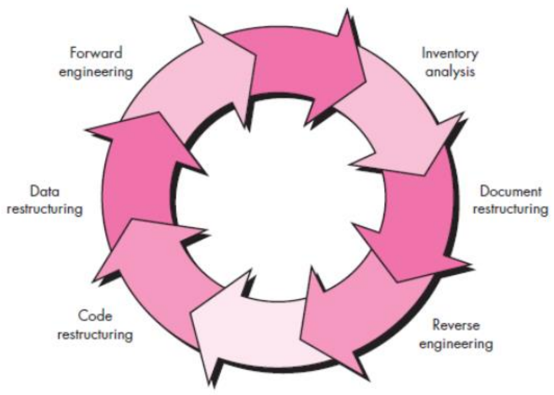
Reverse Engineering

- **What**: Analyzing the software to understand its structure and behavior.
- **Why**: Helps recreate the design from existing code and documentation.
- **Output**: Diagrams that show how the software is built and how it works.

Re-engineering

- **What**: Modifying the software to create a new version.
- **Why**: Improves performance or updates it for new needs.
- **Also Known As**: Renovation or reclamation.

Software Reengineering Process Model



1 **Inventory Analysis**

- **What**: Create a detailed list (like a spreadsheet) of all active software applications.
- **Why**: To know the size, age, and importance of each application.

- **Key Point**: Regularly update this inventory since the status of applications can change.

2. Document Restructuring

- **What**: Improve the documentation of legacy systems, which often have poor documentation.
- **Challenges**:
    - Creating documentation takes a lot of time.
    - Limited resources mean you can only update parts of the documentation that have changed.
- **Goal**: Even if the system is critical, keep documentation concise and essential.

3. Reverse Engineering

- **What**: Analyze existing software (like how a company disassembles a competitor's product) to understand its design and functionality.
- **Tools**: Use special tools to extract information about the software's structure and processes.

4. Code Restructuring

- **What**: Organize and improve the code without changing what it does.
- **Why**: Makes the code cleaner and easier to maintain.

5. Data Restructuring

- **What**: Examine and improve the data architecture of the software.
- **Why**: Poor data architecture makes it hard to adapt and enhance the system.
- **Process**:
    - Analyze current data models.
    - Identify important data objects and attributes.
    - Review data structures for quality.

6. Forward Engineering

- **What**: Use the design information recovered from the software to make improvements.
- **Goal**: Alter or rebuild the system to enhance its overall quality.

Reverse Engineering

- **Definition**: It's the process of analyzing a final product (like software) to recreate its design.

Key Concepts:

1. **Abstraction Level**:
    - Refers to how detailed and sophisticated the design information is that you can get from the source code.
    - Higher abstraction means more complex design insights.
2. **Completeness**:
    - Indicates how detailed the information is at that level of abstraction.
    - A complete reverse engineering process gives you a thorough understanding of the system's design.
3. **Interactivity**:
    - This is about how well humans work with automated tools during the reverse engineering process.
    - Better integration can lead to more effective outcomes.
4. **Directionality**:
    - **One-way**: This is focused on maintenance activities, where you extract information to fix or maintain the software.
    - **Two-way**: Involves both understanding the existing design and using that information to restructure or improve the software.

Software Maintenance

- **Definition**: This is the process of making changes to a software system after it has been delivered.
- **Purpose**: Changes can be for fixing errors, improving design, or adding new features based on user needs.

Types of Software Maintenance:

1. **Fault Repairs**:
    - **What**: Fixing errors in the code or design.
    - **Cost**:
        - **Coding Errors**: Cheap to fix.
        - **Design Errors**: More expensive since they might require rewriting parts of the program.
        - **Requirements Errors**: Most expensive, often needing a complete redesign of the system.
2. **Environmental Adaptation**:
    - **What**: Adjusting the software when something in its environment changes.
    - **Examples**: Changes in hardware, operating systems, or supporting software.

    - **Need**: The software must be updated to work with these new environments.

3. **Functionality Addition**:
    - **What**: Adding new features or changing system requirements.
    - **Scale**: Usually involves more extensive changes compared to fixing errors.

Other Types of Maintenance:

- **Corrective Maintenance**: This term is commonly used for fixing faults.
- **Adaptive Maintenance**: Refers to modifying the software to fit new environmental conditions.
- **Perfective Maintenance**: Means enhancing the software by adding new features or improving performance.

Unit testing :- Unit testing is a software testing method where individual components or modules of the HealthCare Management System are tested in isolation to ensure that each unit functions correctly according to its specifications. The goal is to validate that each unit of the code performs as intended, identifying any defects early in the development process

System and Integration Testing :- is a critical phase in the software testing lifecycle where the entire HealthCare Management System is evaluated as a complete and integrated solution. This testing ensures that all components and modules work together as intended and that the system meets its specified requirements. System testing focuses on validating the end-to-end functionality, performance, and compliance of the entire system, while integration testing assesses the interactions between individual modules and external systems.

Performance Testing :- Evaluates the responsiveness, speed, scalability, and stability of the HealthCare Management System under various conditions. **Stress Testing**, a subset of performance testing, specifically assesses how the system behaves under extreme conditions, such as high user loads or resource depletion. The goal is to identify the system's breaking point, understand its behavior under stress, and ensure it can recover gracefully from failure.

User Acceptance Testing :- User Acceptance Testing (UAT) is the final phase of testing conducted to ensure that the HealthCare Management System meets the business requirements and is ready for operational use. During UAT, end-users (customers) assess the system against its initial requirements, verifying that it performs as expected in real-world scenarios. This testing aims to validate usability, functionality, and overall satisfaction before deployment.

Batch Testing :- refers to the process of testing a group of functionalities or transactions within the HealthCare Management System as a single unit. This method is particularly useful for validating scenarios that involve bulk operations, such as processing multiple medication orders, managing appointments in bulk, or updating records en masse. The goal of batch testing is to ensure that the system can handle large volumes of data efficiently and accurately while maintaining performance and functionality.

Automated Regression Testing :- Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still works as specified in the requirements.

Beta Testing :- It is a phase of software testing in which a pre-release version of the application is distributed to a select group of end-users outside the development team. The primary goal is to gather real-world feedback on the software's functionality, usability, and overall performance. This phase helps identify any remaining bugs, usability issues, or areas for improvement before the official launch, ensuring the final product meets user expectations and requirements.