SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 2

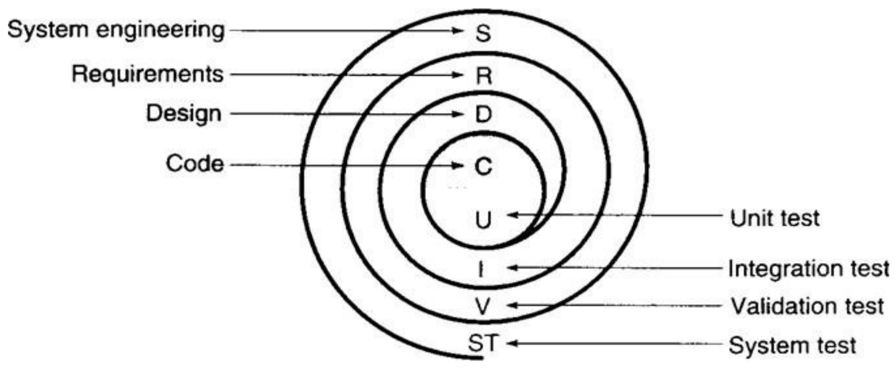
TESTING STRATEGY AND ART OF DEBUGGING

SOFTWARE TESTING STRATEGY FOR CONVENTIONAL SOFTWARE:

A software testing strategy for conventional software ensures the product is high-quality, works as expected, and meets requirements. The process is systematic and follows these steps:

1. **Understand Requirements:** First, the team carefully reviews the software requirements to know exactly what needs to be tested.
2. **Create a Test Plan:** A plan is prepared, explaining what to test, how to test, and the resources needed, including time and people.
3. **Design Test Cases:** Detailed test cases are created to check all possible scenarios, including normal and unusual situations.
4. **Set Up a Test Environment:** A setup is created that is similar to the actual environment where the software will run.
5. **Execute Tests:** Different types of testing, such as checking functionality, performance, and security, are performed to find any issues.
6. **Track and Fix Issues:** Any defects found are recorded, fixed, and tested again to ensure the fixes work and don't create new problems.
7. **Do User Testing:** The software is tested by real users to ensure it meets their needs and works as expected in real-life situations.
8. **Document Everything:** All test plans, test cases, and results are recorded for future reference.
9. **Create a Summary Report:** A report is prepared at the end to summarize the testing activities and results.
10. **Focus on Improvement:** Feedback is gathered to improve the testing process for future projects.

A strategy for software testing may be viewed in the context of the spiral as shown below

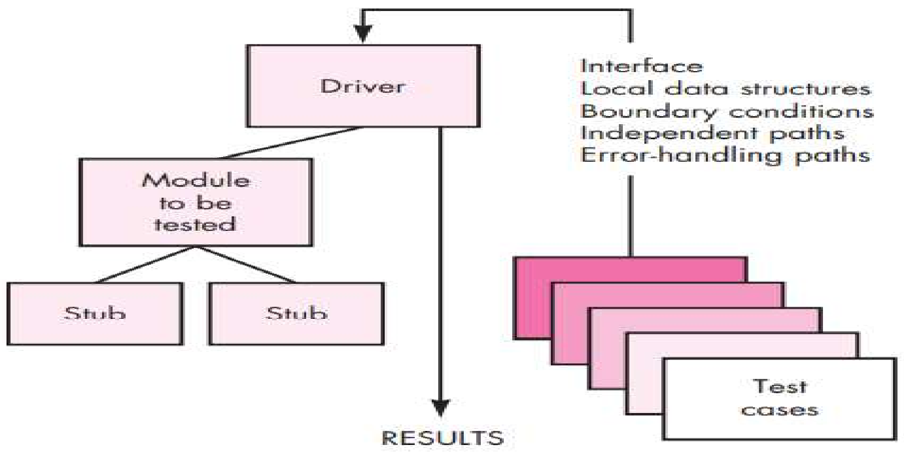


1. Unit Testing

Unit testing is the first step in software testing, focusing on checking individual pieces or "units" of the software in isolation. It’s like testing each brick before constructing a wall. This process ensures that every small part of the software works correctly on its own, helping to catch errors early and making them easier and less expensive to fix.

Tools Used in Unit Testing

In software development, different parts of the system often rely on one another. During unit testing, some of these parts may not be ready. To address this, two tools—**drivers** and **stubs**—are used. These tools allow testing of individual components without needing the entire system to be complete.



1. **Drivers**
   Drivers are small programs that trigger and control the testing of specific software parts. They act as substitutes for the part of the system that normally calls the function being tested.
   - **Example**: Consider testing the login function of a website. Normally, the function is activated when a user clicks the "Login" button. Instead of waiting for the complete user interface to be ready, a driver directly calls the login function with sample inputs (e.g., a username and password) to check if it works properly.
2. **Stubs**
   Stubs are placeholders for incomplete or missing parts of the system. They simulate the behavior of the unavailable components by providing fake responses.
   - **Example**: Think about testing an online store's checkout system. The checkout might rely on a payment gateway that hasn't been connected yet. A stub can act as the payment gateway and provide fake responses like "Payment Successful" or "Payment Failed," allowing the checkout system to be tested without the actual gateway.

How Drivers and Stubs Work Together

Drivers and stubs work as temporary tools to fill the gaps during testing:

- The **driver** starts the test by sending inputs to the unit being tested.
- If the unit relies on other parts of the software that are not yet available, **stubs** step in to mimic those parts and provide fake responses.

This setup ensures that each component can be tested in isolation, even if the other parts are still under development.

Importance of Unit Testing

- **Speeds Up Testing**: Drivers and stubs enable testing of specific features without waiting for the entire system to be ready.
- **Early Detection of Errors**: Testing smaller parts first helps find and fix problems early.

- **Saves Time and Effort**: Isolated testing avoids delays caused by missing or incomplete components.

Unit testing ensures that the smallest building blocks of software work correctly. Once all units function properly, the next stage is to test how these units interact with one another—this is called **integration testing**.

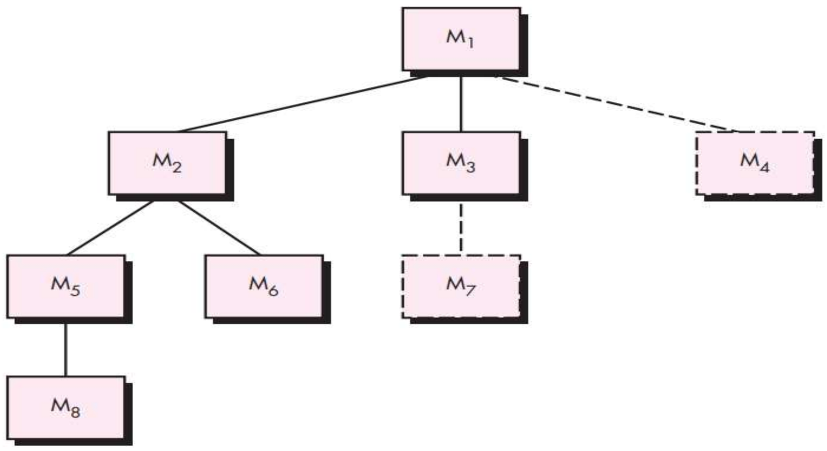
Once individual units are tested, the next step is to test how they interact with one another.

2. Integration Testing

Integration testing ensures that different parts (modules) of the software work together smoothly. After individual units are tested, they are combined, and their interactions are tested to confirm that the system functions as expected when all parts are integrated.

Top-Down Integration Testing

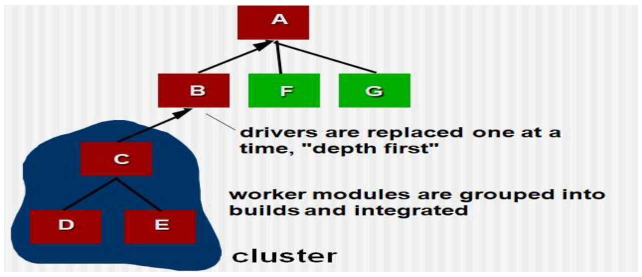
This method begins testing with the main module at the top of the system and progresses downward to the lower modules. Missing or unfinished lower modules are replaced with stubs during testing.



- For example, in a mobile app, the top-level feature, such as the dashboard, is tested first while stubs simulate other sections like notifications or settings.
- In the diagram, testing starts with `M1` at the top, then integrates `M2` and `M3`, while missing modules like `M4` are temporarily replaced with stubs.

Bottom-Up Integration Testing

This method starts testing from the lowest modules and builds upward toward the main module. Higher-level modules not yet available are replaced with drivers that simulate their behavior.


A
B
F
G
C
D
E
drivers are replaced one at a time, "depth first"
worker modules are grouped into builds and integrated
cluster

- For instance, in an e-commerce app, lower-level modules like adding items to the cart are tested first, while a driver simulates the checkout process.
- In the diagram, lower modules `D` and `E` are grouped and tested first. Drivers act as temporary placeholders for higher modules like `B` and `A`, progressing step-by-step until the entire system is integrated.

Stubs and Drivers in Integration Testing

- **Stubs** simulate unfinished modules, allowing testing of higher-level components before all parts are ready. For example, a stub can mimic a payment system for testing the checkout feature.
- **Drivers** simulate higher-level modules to test lower-level components. For example, a driver could mimic user interactions to test data-processing modules.

Integration testing ensures that the system works as a whole, whether built from the top down or the bottom up.

After integration testing confirms that components interact correctly, the entire system is tested as a whole.

3. System Testing

System testing checks the entire software system to ensure it works as a whole. It's like inspecting a fully assembled car to ensure all parts function together.

Why System Testing?

- Ensures the software is complete and ready for users.
- Verifies it meets all requirements.

Types of System Testing:

1. **Functional Testing**: Does it do what it's supposed to do?
   - **Example**: Testing if users can add and remove items from their shopping cart.
2. **Performance Testing**: Can it handle heavy use or high traffic?
   - **Example**: Checking if a website can handle 1,000 users at the same time.
3. **Security Testing**: Is it safe from unauthorized access or hacking?
   - **Example**: Testing if passwords are encrypted.
4. **Compatibility Testing**: Does it work on different devices and browsers?
   - **Example**: Checking if a mobile app runs smoothly on both iPhones and Android phones.

4. Validation Testing

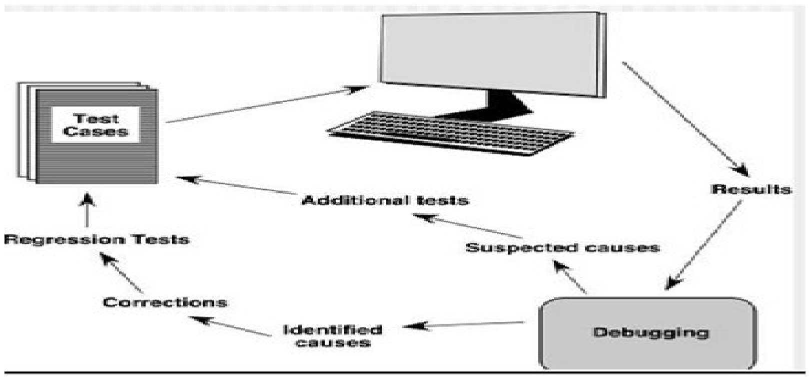
Validation testing checks if the software fulfils user needs and works in real-world conditions.

Types of Validation Testing:

1. **Alpha Testing**:
   - Conducted by the development team in a controlled environment.
   - **Example**: Testing a banking app on office computers.
2. **Beta Testing**:
   - Conducted by real users in their own environments.
   - **Example**: Releasing a beta version of an app to collect user feedback.

THE ART OF DEBUGGING

Debugging is a crucial process in software development where developers identify and fix errors (bugs) in the software. It goes beyond just finding the bug; it involves understanding the root cause of the issue and resolving it without introducing new problems. Debugging ensures the software works as intended and is free from errors.



Steps in Debugging

1. **Identify the Problem**:
   - Understand the issue by reproducing it.
   - Example: If a webpage crashes when submitting a form, try submitting the same form with similar inputs to see the error again.
2. **Locate the Source**:
   - Use tools or manual analysis to find where the problem occurs in the code.
   - Example: If a login form fails, check the function that verifies the username and password.
3. **Analyze the Cause**:
   - Study the code and logic to figure out why the problem happens.
   - Example: A calculation error could be due to incorrect variable initialization or a missing condition in the logic.
4. **Fix the Issue**:
   - Modify the code to resolve the problem.
   - Example: If a variable isn't updated correctly, ensure the correct value is assigned at the right place.
5. **Test the Fix**:
   - Run the software again to confirm the issue is resolved and no new issues are introduced.
   - Example: After fixing a form submission error, test the form with different inputs to ensure it works for all cases.

Types of Debugging / Debugging Strategies

1. **Brute Force Debugging**:
   - Test all possible inputs and conditions until the problem is identified.
   - **Example**: If a function fails for some inputs, feed every possible input into the function until you find which one causes the issue.
2. **Backtracking**:
   - Start at the point where the problem is visible and work backward through the code to find the root cause.
   - **Example**: If a value is incorrect at the end of a calculation, trace each step of the calculation to find where the error occurred.
3. **Cause Elimination**:
   - Systematically disable or isolate parts of the code to pinpoint where the error is happening.
   - **Example**: Temporarily disable different sections of a function to check which section causes the issue.
4. **Print Statement Debugging**:
   - Insert print statements in the code to display variable values or program states at specific points.
   - **Example**: Print the value of a variable inside a loop to see how it changes with each iteration.
5. **Using Debugging Tools**:
   - Tools like debuggers, breakpoints, and logs can simplify the debugging process.
   - **Example**: Use the debugger in an IDE to pause the program at a specific line and inspect all variable values.

Real-Life Example of Debugging

Scenario: A calculator app is giving incorrect results when dividing two numbers.

Steps to Debug:

1. **Identify the Problem**:
   - Observe that the division feature is not working correctly. For example, when dividing 10 by 2, the result is shown as 0 instead of 5.

2. **Locate the Source**:
    - Analyze the code that performs the division. Focus on the function or formula used for the calculation.
3. **Analyze the Cause**:
    - Discover that the division code is written using integer division (`/`) instead of floating-point division, which truncates the result to 0 for non-decimal results.
4. **Fix the Issue**:
    - Modify the code to use the correct division operator (e.g., `float division` instead of `integer division`). Update the function to ensure accurate calculations.
5. **Test the Fix**:
    - Test the calculator by dividing different numbers (e.g., 10/2, 15/3, and 7/2) to ensure the results are correct and accurate (e.g., 5, 5, and 3.5).