Module 1

farisbashatm@gmail.com

Some Popular Errors

Ariane 5 Rocket Failure (1996)

- **Cause:** Software error in the inertial reference system.
- **Details:** Conversion of a 64-bit floating point to a 16-bit signed integer caused an overflow.
- **Impact:** Rocket self-destructed 37 seconds after launch, resulting in a loss of approximately $370 million.
- **Development Context:** Error inherited from Ariane 4, unfit for Ariane 5's dynamics.
- **Preventive Measure:** Led to better code reviews and stricter software testing.

Therac-25 Radiation Machine (1985-1987)

- **Cause:** Software bugs in the control system.
- **Details:** Concurrent programming errors and inadequate safety checks led to massive overdoses of radiation.
- **Impact:** Six known incidents of severe radiation overdoses, resulting in serious injuries and three deaths.
- **Lack of Hardware Interlocks:** Removal of safety interlocks worsened the problem.
- **Regulatory Changes:** Incidents prompted stricter medical device testing and software verification.

Intel Pentium Floating Point Division Bug (1994)

- **Cause:** Flawed division algorithm in the floating-point unit.
- **Details:** Errors in the lookup table used for division operations caused incorrect results for certain calculations.
- **Impact:** Loss of public trust and financial cost to Intel estimated at $475 million due to recall and replacement of affected processors.
- **Detection and Disclosure:** Discovered by a professor; publicized issue led to transparency focus.

What is Software Testing?

- Software testing ensures that a program behaves as expected.
- It involves running the program with various inputs to check its behavior.

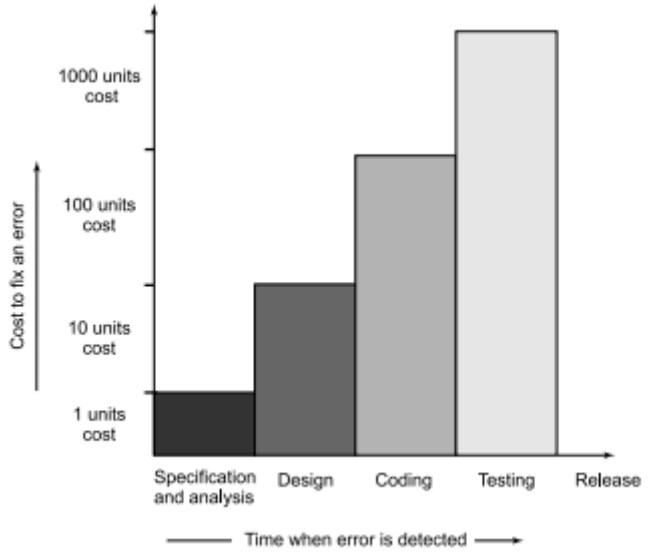
- Test cases are created to cover different scenarios and functionalities.
- Testing helps identify bugs and ensures the reliability and quality of the software.
- A simple Python example:

```
def add(a, b):
    return a + b

# Testing the add function
result = add(2, 3)
expected_result = 5
if result == expected_result:
    print("Addition function passed the test!")
else:
    print("Addition function failed")
```

This example tests the addition function by comparing the result with an expected value. If they match, the test passes; otherwise, it fails.

Why should we test?



The above graph **Phase wise cost of fixing an error**

Software testing is a very expensive and critical activity; but releasing the software without testing is definitely more expensive and dangerous. No one would like to do it. It is like running a car without brakes.

Goals of Testing Software

Understanding Testing Goals

- **Confusion in Objectives:** Many software engineers are unclear if testing is meant to show correctness, find problems, or achieve something else.

- **Importance of Clarity:** Clear goals in testing help define processes and expected outcomes.

Beizer's Test Process Maturity Levels

- **Level 0:** No distinction between testing and debugging. Common among inexperienced programmers.
- **Level 1:** Testing aims to show correctness, which is often impractical for complex software.
- **Level 2:** Focus on finding failures, which can create an adversarial relationship between testers and developers.
- **Level 3:** Testing aims to reduce the risk of using software, fostering collaboration between testers and developers.
- **Level 4:** Testing as a discipline to enhance software quality, with testers leading in quality improvement and training developers.

Software Testing Terminologies

Verification

- **Definition:** Ensuring that products of a development phase meet the requirements of the previous phase.
- **Example:** Checking if the design documents correctly implement the specifications.
- **Focus:** Technical aspects, ensuring specifications are met.

Validation

- **Definition:** Evaluating software at the end of development to ensure it meets user needs and intended use.
- **Example:** Conducting user acceptance testing to confirm the software meets user expectations.
- **Focus:** User satisfaction and domain-specific requirements.

Testing

- **Definition:** Executing software to find defects.
- **Example:** Running the application to check if all buttons function correctly.
- **Purpose:** Identify issues, ensure software functions as intended.

Faults, Errors, and Bugs

- **Fault:** A flaw in the software that can cause a failure.
  - **Example:** A piece of code that does not handle null inputs correctly.
- **Error:** Human action or oversight that produces incorrect results.

    - **Example:** A developer incorrectly implements a calculation formula.
- **Bug:** A manifestation of a fault in the software.
    - **Example:** An application crashes when a user inputs a specific character.

Test Cases

- **Definition:** Specific conditions or inputs used to test the software.
- **Example:** A test case to check if a login function works with correct and incorrect passwords.

Coverage Criteria

- **Definition:** Measures used to determine the extent to which the software has been tested.
- **Example:** Statement coverage ensures every line of code is executed at least once during testing.

Types of Testing

Unit Testing

- **Definition:** Testing individual components, such as functions or methods, in isolation.
- **Example:** A developer tests a single function that calculates the sum of two numbers.
- **Performed By:** Programmers during the development phase.
- **Focus:** Ensuring that each unit performs as expected independently.
- **Tools:** JUnit, NUnit, PyTest.

Integration Testing

- **Definition:** Testing combined parts of an application to ensure they work together.
- **Example:** Checking if data flows correctly between a login module and a user dashboard.
- **Performed By:** Developers and integration test engineers.
- **Focus:** Detecting interface and interaction errors between modules.
- **Types:** Big Bang, Top-Down, Bottom-Up, and Incremental integration testing.

System Testing

- **Definition:** Testing the complete and integrated software system to verify it meets requirements.
- **Example:** Testing an entire e-commerce application for functionality, security, and performance.
- **Performed By:** QA teams before the product is delivered to the customer.

- **Focus:** Validating end-to-end system specifications.
- **Activities:** Creating test plans, designing test suites, and preparing test environments.

Acceptance Testing

- **Definition:** Testing conducted to determine if the software meets the customer's requirements.
- **Example:** A client tests an inventory management system to ensure it fulfills their business needs.
- **Performed By:** Customers or end-users.
- **Focus:** Meeting business needs and user expectations.
- **Types:** User Acceptance Testing (UAT) and Business Acceptance Testing (BAT).

Beta Testing

- **Definition:** Testing by end-users in a real-world environment before the final release.
- **Example:** Releasing a mobile app to a group of users to identify any issues before the public launch.
- **Performed By:** Selected end-users.
- **Focus:** Gathering real-world feedback and uncovering bugs not found in internal testing.
- **Advantages:** Helps ensure the software works well under real conditions and meets user expectations.

Functional Testing

- **Definition:** Testing to verify that each function of the software operates according to the requirements.
- **Example:** Ensuring a search feature returns correct results.
- **Performed By:** QA teams.
- **Focus:** Validating actions and operations of the system against requirements.
- **Types:** Smoke testing, sanity testing, and regression testing.

Stress Testing

- **Definition:** Testing to evaluate how the software performs under extreme conditions.
- **Example:** Simulating high traffic on a website to see how it handles peak loads.
- **Performed By:** Performance engineers.
- **Focus:** Assessing the robustness and stability of the system under stress.
- **Outcome:** Identifies potential failures and bottlenecks under extreme conditions.

Performance Testing

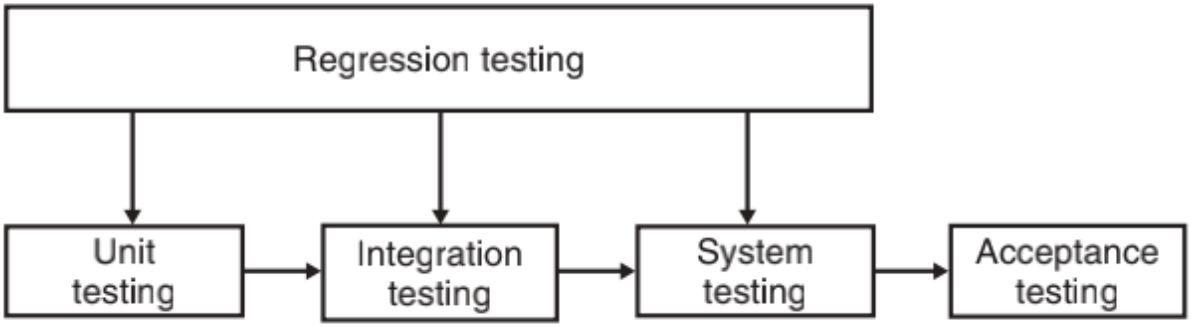
- **Definition:** Testing to assess the speed, responsiveness, and stability of a system under a given workload.
- **Example:** Measuring how quickly a webpage loads under various network conditions.
- **Performed By:** Performance engineers.
- **Focus:** Evaluating system performance under expected workloads.
- **Types:** Load testing, stress testing, and endurance testing.

Usability Testing

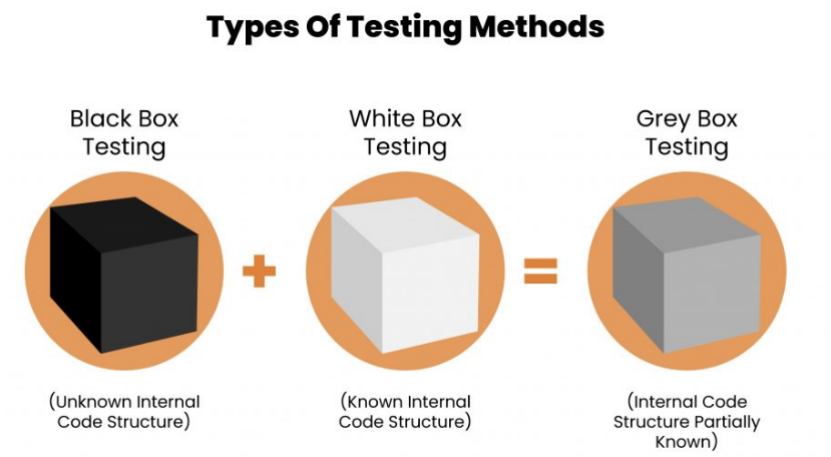
- **Definition:** Testing to determine how user-friendly and intuitive the software is.
- **Example:** Observing users navigate through an application to identify usability issues.
- **Performed By:** UX researchers.
- **Focus:** Enhancing user satisfaction by improving usability.
- **Methods:** User observations, surveys, and task completion assessments.

Regression Testing

- **Definition:** Re-running previously conducted tests to ensure existing functionality is not broken by new changes.
- **Example:** Re-testing a login feature after adding a new password recovery option.
- **Performed By:** QA teams throughout the development lifecycle.
- **Focus:** Ensuring new updates do not negatively affect existing functionality.
- **Strategies:** Selective regression testing, full regression testing, and automated regression testing.



Testing Methods

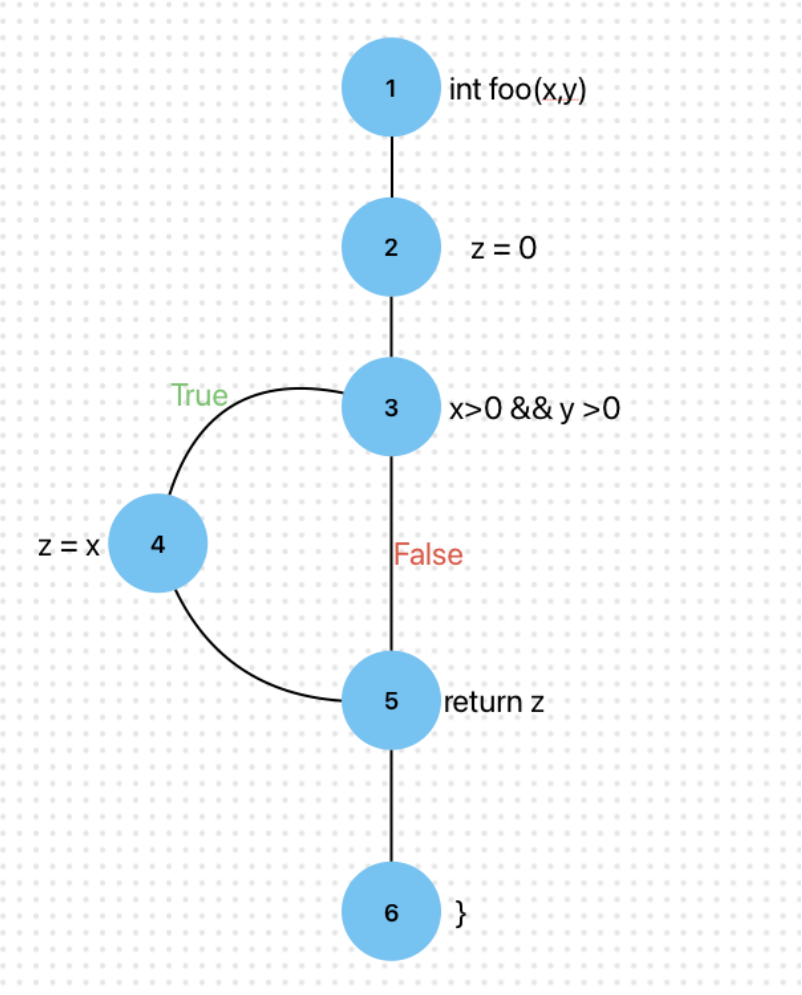


Aspect	Black-box Testing	White-box Testing	Grey-box Testing
Principle	Focuses on external behavior without examining internal logic.	Tests internal logic and structure of the software directly.	Combines elements of both black-box and white-box approaches.
Level of Knowledge	Tester has no knowledge of internal code or structure.	Tester has full access to internal code and design.	Tester has partial knowledge of internal code and structure.
Techniques	Equivalence partitioning, boundary value analysis, scenario-based testing.	Coverage criteria: statement, branch, and path coverage.	Combination of black-box and white-box techniques.
Test Design	Based on specifications and requirements.	Based on code structure and implementation details.	Combines specification-based and structure-based testing.
Dependency	Independent of code changes.	Highly dependent on code changes.	Moderately dependent on code changes.
Scope	Suitable for system and acceptance testing.	Commonly used in unit and integration testing.	Useful for integration and system testing.
Visibility	Focuses on user interactions and inputs/outputs.	Focuses on code paths and internal data structures.	Balances external behavior and internal structure.

Coverage Criteria

Coverage criteria are metrics used to measure the extent to which the code has been executed by a set of test cases. The below example code is for explaining different coverage criteria :

```
1. int foo(int x, int y) {
2.     int z = 0;
3.     if ((x > 0) && (y > 0)) {
4.         z = x;}
5.     return z;
6. }
```



Statement Coverage

- **Definition:** Ensures every statement in the code is executed at least once.
- **Example:** For `foo(1, 1)`, the statements executed are:
  - `int z = 0;`
  - `if ((x > 0) && (y > 0)) {`
  - `z = x; }`
  - `return z;`
- **Objective:** Verify that all lines of code are tested.

Branch Coverage

- **Definition:** Ensures every branch (true/false) of each decision point is executed.
- **Example:** For `foo(1, 1)` and `foo(-1, -1)`, the branches covered are:

    - `(x > 0) && (y > 0)` evaluates to true.
    - `(x > 0) && (y > 0)` evaluates to false.
- **Objective:** Verify that each decision branch is tested.

Condition Coverage

- **Definition:** Ensures each individual condition in a decision is tested for both true and false outcomes.
- **Example:** For `foo(1, 1)`, `foo(1, -1)`, `foo(-1, 1)`, and `foo(-1, -1)`:
    - `x > 0` is true/false.
    - `y > 0` is true/false.
- **Objective:** Validate each condition within a decision.

Path Coverage

- **Definition:** Ensures every possible path through the code is executed.
- **Example:** For `foo(1, 1)`, `foo(-1, -1)`, `foo(1, -1)`, `foo(-1, 1)`:
    - Path 1: `if` condition true, `z = x`.
    - Path 2: `if` condition false.
- **Objective:** Test all potential execution paths through the program.

Function Coverage

- **Definition:** Ensures every function in the code is called at least once.
- **Example:** Testing `foo` with various inputs ensures the function is called.
- **Objective:** Validate that all functions are tested.

Applying Coverage Criteria to Example Code

Example Code Analysis

```
int foo(int x, int y) {
    int z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    }
    return z;
}
```

1. **Statement Coverage:**
    - Test Case 1: `foo(1, 1)` covers all statements.
    - Test Case 2: `foo(-1, -1)` ensures the return statement is also executed without entering the `if`.
2. **Branch Coverage:**

- Test Case 1: `foo(1, 1)` for true branch.
- Test Case 2: `foo(-1, -1)` for false branch.

3. **Condition Coverage:**
   - Test Case 1: `foo(1, 1)` where both conditions are true.
   - Test Case 2: `foo(1, -1)` where the first condition is true, and the second is false.
   - Test Case 3: `foo(-1, 1)` where the first condition is false, and the second is true.
   - Test Case 4: `foo(-1, -1)` where both conditions are false.
4. **Path Coverage:**
   - Path 1: Entering the `if` and executing `z = x` (`foo(1, 1)`).
   - Path 2: Skipping the `if` and directly returning `z` (`foo(-1, -1)`).
5. **Function Coverage:**
   - Test Case: Any valid input to `foo`, e.g., `foo(1, 1)` or `foo(-1, -1)`, ensures the function is called.

By covering these criteria, you can ensure a more thorough validation of the code's behavior under various conditions, leading to higher software quality and reliability.

Sample Test Case for ATM: Withdrawal Transaction

Test Case Title: Withdrawal Transaction - Valid Amount

Description: This test case verifies the functionality of withdrawing cash from the ATM with a valid withdrawal amount.

Preconditions:

1. The ATM machine is powered on and operational.
2. The user has inserted a valid ATM card and entered their correct PIN.
3. The user has navigated to the withdrawal transaction option.

Test Steps:

1. Press the "Withdraw" button on the ATM interface.
2. Select the desired withdrawal amount by pressing the corresponding buttons (e.g., B2 for $40 or B4 for $100).
3. Press the "OK" button to confirm the withdrawal.

Expected Results:

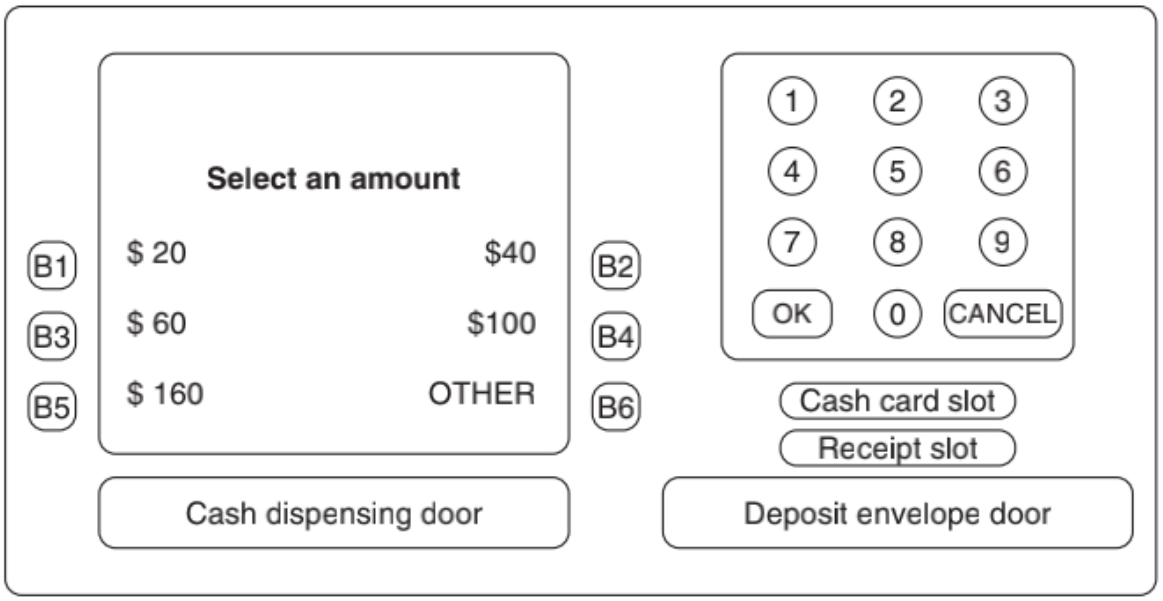
- The ATM dispenses the requested amount of cash.
- The user's account balance is updated accordingly.
- A receipt is printed with details of the transaction, including the remaining balance.

Postconditions:

1. The user receives the requested cash.
2. The user's account balance is accurately updated.
3. The user retrieves their ATM card and any printed receipt.

Notes:

- Additional test cases may include scenarios for insufficient funds, invalid PIN, and transaction timeouts.



Positive Test Cases:

1. **Valid Withdrawal:**
   - Preconditions: ATM operational, valid card & PIN.
   - Steps: Select Withdraw, choose amount, confirm.
   - Expected: Cash dispensed, receipt printed.
2. **Valid Deposit:**
   - Preconditions: ATM operational, valid card & PIN.
   - Steps: Select Deposit, insert cash/check, confirm.
   - Expected: Deposit accepted, receipt printed.
3. **Check Balance:**
   - Preconditions: ATM operational, valid card & PIN.
   - Steps: Select Check Balance.
   - Expected: Display current balance.
4. **Fund Transfer:**
   - Preconditions: ATM operational, valid card & PIN, sufficient balance.
   - Steps: Select Fund Transfer, enter recipient & amount, confirm.
   - Expected: Transfer completed, receipt printed.

Negative Test Cases:

1. **Insufficient Funds:**
   - Preconditions: ATM operational, valid card & PIN, insufficient balance.
   - Steps: Attempt withdrawal or transfer.
   - Expected: Error message, no transaction.
2. **Invalid PIN:**
   - Preconditions: ATM operational, valid card.
   - Steps: Enter invalid PIN.
   - Expected: Card locked after retries.
3. **Invalid Transaction:**
   - Preconditions: ATM operational, valid card & PIN.
   - Steps: Attempt unsupported transaction.
   - Expected: Error message, no transaction.
4. **Card Retention:**
   - Preconditions: ATM operational, valid card & PIN.
   - Steps: Fail to retrieve card after transaction.
   - Expected: ATM retains card for security.

Different Levels of Software testing

Testing in software development occurs at multiple levels throughout the product's lifecycle, illustrated by the classical V model.

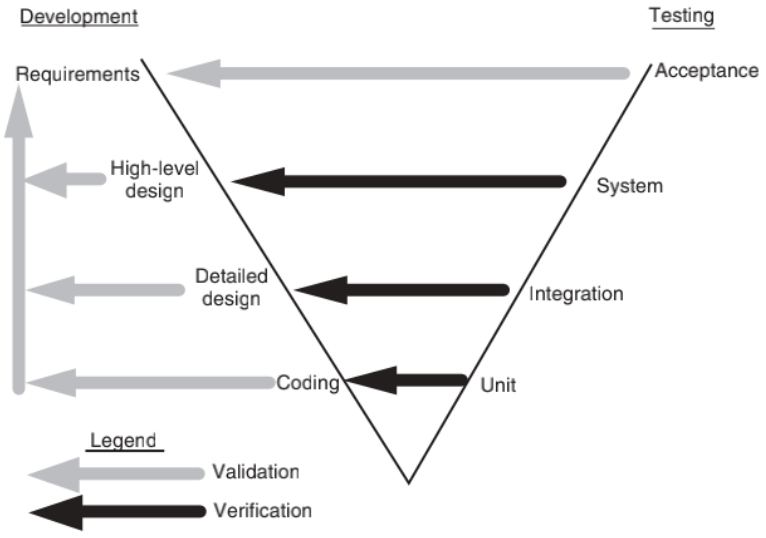


Figure 1.7 Development and testing phases in the V model.

Software testing is conducted at various levels, each with distinct objectives:

1. **Unit Testing**
   - **Objective**: Verify individual units of code.
   - **Scope**: Smallest testable parts of software.
   - **Performed by**: Developers.

- **Techniques**: Functional and structural testing.
- **Challenges**: Requires stubs and drivers to test units in isolation.

2. **Integration Testing**
   - **Objective**: Test the interaction between integrated units.
   - **Scope**: Combined units/modules.
   - **Performed by**: Testers.
   - **Focus**: Interface issues and coupling between units.
   - **Strategies**: Top-down, bottom-up, and sandwich approaches.
3. **System Testing**
   - **Objective**: Validate the complete integrated system.
   - **Scope**: Entire system in a simulated environment.
   - **Performed by**: Testers under a test team leader's supervision.
   - **Techniques**: Mainly functional, some structural.
   - **Focus**: Both functional and non-functional requirements (e.g., performance, security).
4. **Acceptance Testing**
   - **Objective**: Confirm the system meets business requirements.
   - **Scope**: Final product.
   - **Performed by**: Customers or end-users.
   - **Types**: Alpha (internal), Beta (external users).
   - **Location**: Typically at the customer’s site.

Each level ensures the software's functionality, integration, and performance align with user expectations and requirements.

Testing Activities

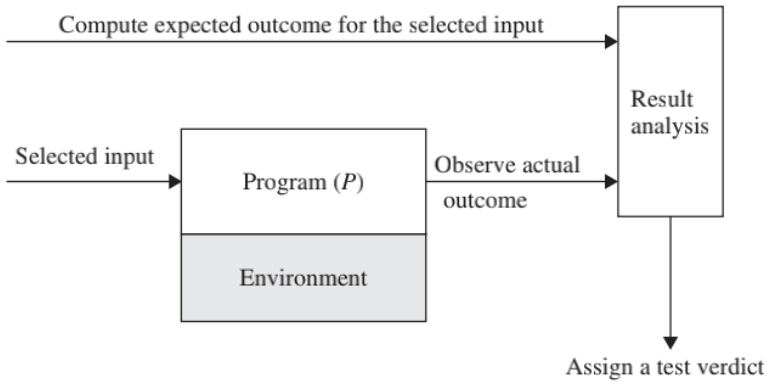


Figure 1.6 Different activities in program testing.

Testing a program involves several structured activities to ensure comprehensive evaluation:

1. **Identify an Objective:**

- Define a clear purpose for the test case to ensure the program meets specific requirements.

2. **Select Inputs:**
   - Choose test inputs based on the test objective, requirements, source code, or expected usage.
3. **Compute Expected Outcome:**
   - Determine the expected result based on the program's specifications and the selected inputs.
4. **Set up the Execution Environment:**
   - Prepare necessary systems and conditions (e.g., network connections, databases) for the program's execution.
5. **Execute the Program:**
   - Run the program with the selected inputs and observe the actual outcomes.
6. **Analyze the Test Result:**
   - Compare actual outcomes with expected outcomes to assign a test verdict:
     - **Pass:** Expected outcome achieved.
     - **Fail:** Expected outcome not achieved.
     - **Inconclusive:** Results unclear, requiring further testing.

At last **Write a Test Report** which is a document how to reproduce failures, analyze issues, and provide details on inputs, expected outcomes, and the execution environment.
This process ensures thorough testing and clear documentation for any identified faults.

Software Quality

Software quality is a complex, context-dependent concept with different interpretations across various domains. Kitchenham and Pfleeger outline five key views of quality:

1. **Transcendental View:**
   - Quality is recognized but difficult to define, much like recognizing art or beauty.
2. **User View:**
   - Quality is fitness for purpose, meeting user needs and expectations.
3. **Manufacturing View:**
   - Quality is conformance to specifications, ensuring the product meets predefined criteria.
4. **Product View:**
   - Quality is based on inherent product characteristics that determine its overall performance and reliability.
5. **Value-Based View:**

- Quality is measured by how much customers are willing to pay, indicating its perceived value.