SOFTWARE TESTING

Prepared By: Natabar Khatri

New Summit College

Unit 8

INTRODUCTION

- Software testing is a critical component of software engineering aimed at achieving two primary, **interconnected goals**:
  - **Demonstrate Correctness:** To show developers and customers that the software meets its specified requirements and behaves as intended.
  - **Discover Defects:** To find inputs or scenarios that cause the software to behave incorrectly, undesirably, or in a way that does not conform to its specification. These unexpected behaviors are caused by defects (or "bugs") in the code.
- Testing involves executing a program with artificial data (test data) and checking the results for errors, anomalies, or information about non-functional attributes like performance.

VALIDATION AND VERIFICATION TESTING

Verification and Validation (V&V) are overarching processes that encompass testing. They are often confused but have distinct meanings, succinctly defined by Barry Boehm:

- **Validation: "Are we building the right product?"**
  - This is an external process. It ensures the software meets the customer's actual needs and expectations, which may not be perfectly captured in the specification. It's about building something of value for the user.
- **Verification: "Are we building the product right?"**
  - This is an internal process. It checks that the software correctly implements all its specified functional and non-functional requirements. It's about ensuring the development is done correctly.

VALIDATION AND VERIFICATION TESTING

Testing's Role in V&V:

Testing activities contribute to both verification and validation, but two specific testing approaches align closely with these concepts:

- **Validation Testing:** Uses test cases that reflect the system's expected, normal use. The goal is to demonstrate that the system performs correctly under standard conditions.
- **Defect Testing:** Uses test cases deliberately designed to be obscure and challenging to expose defects. These tests need not reflect normal usage; their sole purpose is to "break" the system and reveal hidden bugs.

VALIDATION AND VERIFICATION TESTING

The Input-Output Model:

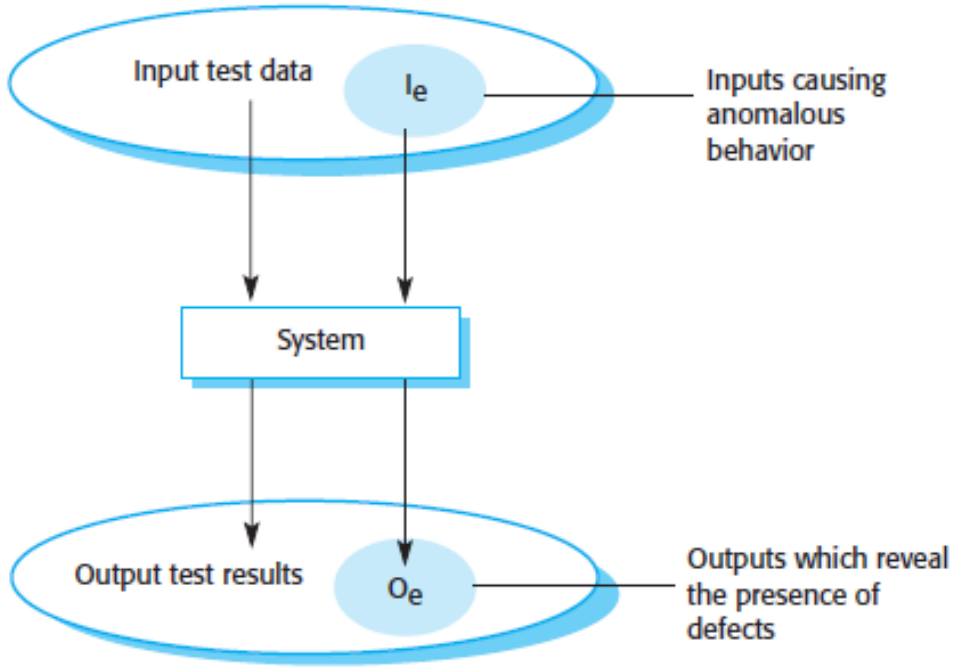

Fig: 8.1: Input-output model of program testing

VALIDATION AND VERIFICATION TESTING

The Input-Output Model:

This models a system as a black box. The key idea is that there is a set of inputs (Ie) that will cause the system to generate erroneous outputs (Oe). The core objective of defect testing is to find as many inputs in Ie as possible.

The Limits of Testing:

It is crucial to understand that testing can never prove the absence of defects. As Edsger Dijkstra stated, "Testing can only show the presence of errors, not their absence." The goal of V&V is instead to establish confidence that the software is "fit for purpose." The required level of confidence depends on:

Software Purpose: Safety-critical systems require much higher confidence.

User Expectations: Users may tolerate more faults in a new, beneficial system.

Marketing Environment: Competitive pressure might lead to releasing a product before it is fully tested

VALIDATION VS VERIFICATION TESTING

Feature	Validation Testing	Defect Testing (Verification)
Core Question	Are we building the right product?	Are we building the product right?
Primary Goal	Demonstrate that the software meets its specified requirements and customer expectations.	Find inputs or sequences that reveal defects, incorrect behavior, or non-conformance to the specification.
Mindset	"Does the system perform correctly when used as we expect it to be used?"	"How can I make this system fail? Where are its weaknesses?"
Test Case Design	Based on **expected use cases and requirements**. Reflects normal, typical, and correct inputs.	Can be **deliberately obscure and unexpected**. Does not need to reflect normal use; aims to probe boundaries and edge cases.
Relationship to V&V	The primary testing activity for the **Validation** process.	A crucial testing activity for the **Verification** process.

SOFTWARE INSPECTION

- Software Inspections are a **static** V&V technique, meaning they are conducted without executing the software. They involve a systematic, peer review of any readable representation of the software, such as requirements documents, design models (UML diagrams), or source code.

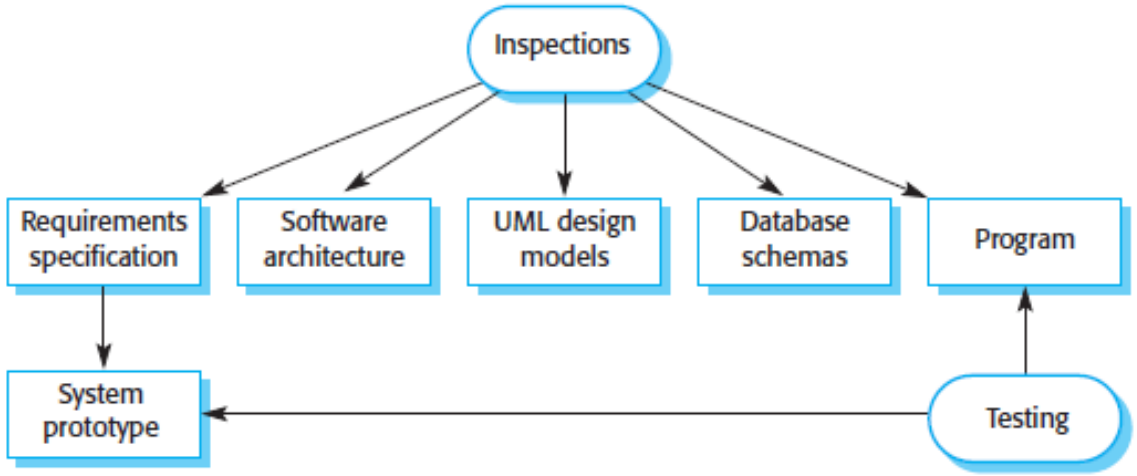

Fig: 8.2: Software Inspection

SOFTWARE INSPECTION

- **Advantages of Inspections over Testing:**
  - **No Error Masking:** During testing, one error can hide others. Since inspections don't involve execution, this interaction is avoided, allowing a single session to discover many errors.
  - **Early and Cost-Effective V&V:** Incomplete versions of the system (e.g., a design document) can be inspected without the need to write complex test harnesses to run code.
  - **Broader Quality Assessment:** Inspections can evaluate qualities beyond functionality, such as adherence to standards, maintainability, portability, and the quality of algorithms and programming style.

- **Effectiveness and Limitations of Software Inspection:**
  Studies have shown inspections to be extremely effective, with some claims of discovering 60-90% of defects. However, they **cannot replace testing**. Inspections are poor at finding defects related to:
  - Unexpected interactions between components.
  - Timing issues and race conditions.
  - System performance problems.

SOFTWARE TESTING PROCESS

- The traditional testing process, as shown in Figure 8.3, involves a structured sequence of activities:
  1. **Design Test Cases:** A test case is a specification that includes the test inputs, the expected outputs, and the purpose of the test.
  2. **Prepare Test Data:** This is the actual set of inputs devised for the test.
  3. **Run Program with Test Data:** The system is executed using the prepared test data.
  4. **Compare Results to Test Cases:** The actual output is compared to the expected output. Discrepancies are reported as potential defects.

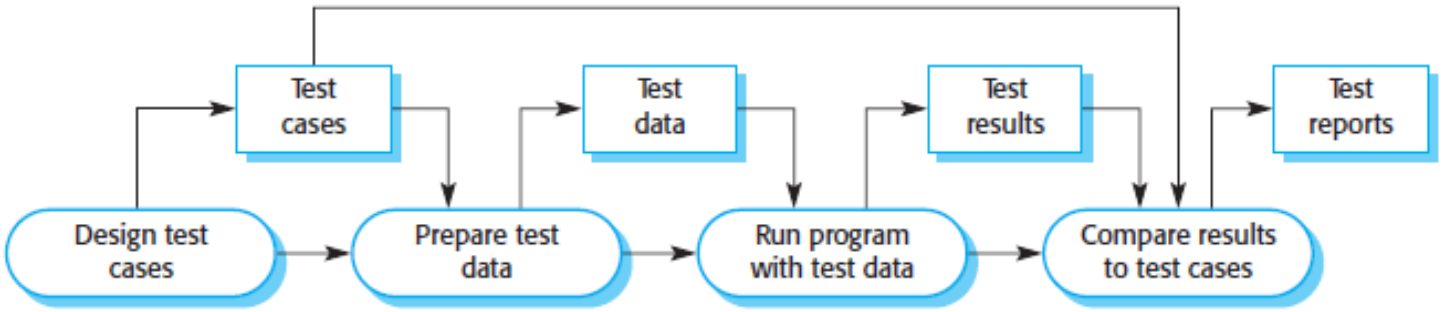

Figure 8.3: a model for software testing process

COMPONENTS OF A TEST CASE

Field	Description
Test Case ID:	Unique identifier for the test case
Test Scenario:	The functionality or module to be tested
Preconditions:	Conditions that must be met before test execution
Test Steps:	Step-by-step instructions to execute the test
Test Data:	Input data to be used
Expected Result:	The expected output after executing the test
Actual Result:	The actual output observed (filled during execution)
Status:	Pass / Fail (based on comparison of expected and actual results)

EXAMPLE: TEST CASE FOR LOGIN FUNCTIONALITY

Let’s consider an example of testing the **Login** feature of a website.

Field	Description
Test Case ID	TC_Login_01
Test Scenario	Verify that user can successfully log in with valid credentials
Preconditions	User must be registered with a valid username and password
Test Steps	1. Open the login page 2. Enter a valid username 3. Enter a valid password 4. Click on the “Login” button Test
Data	username: admin Password: admin@123
Expected Result	The user should be redirected to the Dashboard page
Actual Result	To be filled after execution
Status	Pass / Fail

SOFTWARE TESTING PROCESS

- **Levels of Testing:**
  A commercial system typically undergoes three stages of testing:

  1. **Development Testing:** Conducted by the developers themselves during coding to discover and fix bugs early.
  2. **Release Testing:** Conducted by a separate, independent testing team on a complete version. The goal is to verify that the system meets its requirements as specified to the stakeholders.
  3. **User Testing:** Conducted by users in their own environment to validate that the system meets their needs. **Acceptance testing** is a formal type of user testing where the customer decides whether to accept the delivered system.

SOFTWARE TESTING PROCESS

Manual vs. Automated Testing:

- **Manual Testing:** A tester manually executes steps, provides inputs, and observes outputs. It is flexible but slow and prone to human error.
- **Automated Testing:** Test scripts are written to execute tests automatically. This is essential for:
  - **Regression Testing:** Re-running tests to ensure new changes haven't broken existing functionality.
  - **Performance and Load Testing:** Simulating thousands of users.
  - **Limitation:** It cannot judge aesthetic elements (e.g., GUI layout) or easily detect all unanticipated side effects. Test cases must still be designed by humans.

DEVELOPMENT TESTING

- The primary goal of Development Testing is to identify and eliminate bugs before the software is passed to an independent testing team or released to users.
- It encompasses all testing activities carried out by the team developing the system itself.
- A key characteristic of development testing is that the tester is often the programmer who wrote the code. However, some methodologies, like programmer/tester pairs, advocate for a dedicated tester within the development team to design tests and assist in the process.
- For critical systems, a more formal approach is used, involving a separate testing group within the development team that is responsible for developing tests and maintaining detailed records.
- Development testing is inherently a **defect testing process**, focused on discovering bugs.
- The process is typically structured into three distinct but progressive stages:
  - **Unit Testing**
  - **Component Testing**
  - **System Testing**

UNIT TESTING

- **1. Unit Testing**
- Unit testing is the most granular level of testing, focusing on individual program units, such as functions, methods, or object classes.
- **For Object Classes:** Tests should be designed to achieve comprehensive coverage of the object's features. This involves:
  - Testing all operations (methods) associated with the object.
  - Setting and checking the value of all its attributes.
  - Putting the object into all possible states by simulating events that cause state changes.
- **Challenge of Inheritance:** Inheritance complicates unit testing. An operation that works correctly in a parent class may not work as expected in a subclass due to different assumptions about other operations or attributes. Therefore, inherited operations must be tested in every subclass where they are used.

COMPONENT TESTING

- Software components are often composites of several interacting objects. Component testing (or integration testing) builds upon unit testing by integrating individual units to create composite components.
- **Focus:** Testing the **interfaces** that provide access to the component's functions. The assumption is that the individual units within the component have already passed their unit tests.
- **Goal:** To uncover **interface errors** that result from interactions between the objects inside the component. These errors are often not detectable by testing the objects in isolation.

SYSTEM TESTING

- System testing involves integrating some or all of the system's components and testing the system as a whole.
- **Focus:** Testing **component interactions** and the system's **emergent behavior**—functionality and characteristics that only become apparent when components are combined.
- **Key Differences from Component Testing:**
  - It involves integrating reusable, off-the-shelf, or third-party components with newly developed ones.
  - It is a collective process, often involving components from different team members or sub-teams, and may be carried out by a separate testing team.
- System testing must verify both **planned emergent behavior** (e.g., a new feature emerging from the integration of two components) and check for the absence of **unplanned and unwanted behavior**.

TEST DRIVEN DEVELOPMENT (TDD)

- Test-Driven Development (TDD) is a modern software development approach that fundamentally inverts the traditional development sequence.
- Instead of writing code and then creating tests for it, developers using TDD **write tests first** and then implement the code required to make those tests pass.
- This methodology, originally popularized by Extreme Programming (XP), has gained mainstream acceptance and is now used across both agile and plan-driven development processes.

TEST DRIVEN DEVELOPMENT (TDD)

The TDD Process

The fundamental TDD process, often called the "Red-Green-Refactor" cycle, is a disciplined, iterative loop consisting of the following steps:

- **Identify a Small Functional Increment:** The process begins by identifying a tiny, specific piece of functionality to be added. This should be implementable in just a few lines of code.
- **Write a Failing Test (Red):** Before writing any implementation code, the developer writes an automated test for the desired functionality. This test is then run alongside all existing tests. Since the functionality has not been implemented yet, **the new test is expected to fail**. This "Red" state is crucial as it validates that the test is correctly detecting the absence of the feature.
- **Implement the Code (Green):** The developer writes the minimal amount of code necessary to make the failing test pass. The goal at this stage is not to write perfect, elegant code, but simply to achieve a "Green" state where all tests pass.
- **Refactor the Code:** Once the test passes, the developer can now improve the code's structure, readability, and efficiency without changing its external behavior. This involves eliminating duplication, improving names, and simplifying design—all while relying on the test suite to ensure that no functionality is broken during the cleanup.
- **Repeat:** The cycle then repeats for the next small increment of functionality.

RELEASE TESTING

- Release testing is a critical phase in the software development lifecycle where a specific version (or release) of a system is tested to determine if it is ready for use outside the development team.
- This process is a form of **validation checking**, focused on demonstrating that the system meets its requirements and is dependable for normal use.
- Release testing is predominantly a **black-box testing** process. Testers view the system as a "black box," deriving tests solely from the system specification without knowledge of the internal code. This is also known as **functional testing**.
- Three systematic approaches to designing release tests are
  - **requirements-based testing,**
  - **scenario testing, and**
  - **performance testing.**

RELEASE TESTING

1. Requirements-based testing to ensure specification compliance

- This is a systematic approach where each requirement is analyzed to derive a set of tests for it. A fundamental principle is that requirements must be **testable**.
- **Process:** For each requirement, a tester designs one or more test cases to verify that the system's behavior conforms to that requirement.

2. Scenario testing to validate real-world usability and feature integration

- Scenario testing uses realistic, narrative stories about how the system might be used to develop test cases. These scenarios should be credible, complex, and meaningful to stakeholders.
- **Process:** Testers "play out" the scenario, acting as a user would, and observe the system's behavior. This often involves testing combinations of features and even making deliberate mistakes to check error handling.

3. Performance Testing to verify non-functional attributes and robustness under load

- Once a system is fully integrated, performance testing checks emergent properties like speed, scalability, and reliability under load.
- **Process:** This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

USER TESTING: THE ULTIMATE REALITY CHECK

- User Testing is a critical phase in the software testing process where the actual end-users or customers evaluate the system.
- Its primary purpose is to gather direct input and feedback from the people for whom the software is ultimately built.
- This stage is essential because, no matter how thorough internal system and release testing are, they cannot fully replicate the real-world conditions, pressures, and unpredictable workflows of the user's environment.

The Three Types of User Testing

User testing is typically categorized into three distinct types, each serving a different purpose in the development lifecycle.

1. **Alpha Testing**
2. **Beta Testing**
3. **Acceptance Testing**

USER TESTING: THE ULTIMATE REALITY CHECK

1. Alpha Testing

- Alpha testing is an informal, collaborative process where a **selected group of users works closely with the development team** to test early software releases.
- **Purpose:** To identify problems and usability issues that are not apparent to developers, who are often too focused on the technical requirements. Users provide insights into practical workflows that help design more realistic tests.

2. Beta Testing

- Beta testing involves releasing a functional, but potentially unfinished, version of the software to a **larger, external group of users** for evaluation.
- **Purpose:** To discover interaction problems between the software and the vast, unpredictable variety of user hardware, software, and operational environments. It is a form of large-scale, real-world stress testing.

3. Acceptance Testing

- Acceptance testing is a formal, contract-based process where the **customer tests a system to decide if it is ready for deployment** and final payment. It is most common in custom systems development.

ALPHA TESTING VS BETA TESTING

Feature	Alpha Testing	Beta Testing
Primary Goal	To identify major bugs and usability issues **before** the software is seen by a wider audience.	To discover compatibility and reliability issues in **real-world environments** and validate market fit.
Timing & Environment	**Early** in the development cycle. Conducted **internally** in a lab or controlled development environment.	**Later** in the cycle, after alpha testing. Conducted **externally** in the user's own, uncontrolled environment.
Testers	A **selected, small group** of internal users (e.g., QA team, in-house staff) or dedicated early customers working closely with developers.	A **larger, more diverse group** of external users, which can be a selected customer group or a public audience.