**Software testing** is the process of evaluating and verifying that a software application or system works as intended. It helps identify bugs, errors, or gaps in functionality before the product is released to users.

**Approaches to Software Testing**

Software testing ensures that a system meets its requirements and is free of defects. Common approaches include:

**1. Unit Testing**

Tests individual components or functions in isolation.

**2. Integration Testing**

Verifies that different modules or services work together correctly.

**3. System Testing**

Validates the entire system against functional and non-functional requirements.

**4. Acceptance Testing**

Confirms the system meets user expectations and business goals.

**5. Regression Testing**

Ensures new changes haven’t broken existing functionality.

**6. Performance Testing**

Measures responsiveness, stability, and scalability under load.

**7. Security Testing**

Identifies vulnerabilities and ensures data protection.

**8. Black Box Testing**

Tests functionality without knowing internal code structure.

**9. White Box Testing**

Tests internal logic and structure of the code.

These approaches can be combined depending on the project’s complexity and goals.

**What Is Test-Driven Development (TDD)?**

**TDD** is a software development technique where developers write **tests before writing the actual code**. It emphasizes small, repeatable cycles to ensure correctness and drive design.

**Key Concepts:**

* Write a test that defines a desired improvement or new function.
* Run the test and watch it fail (since the feature isn’t implemented yet).
* Write the minimum code to make the test pass.
* Refactor the code for clarity and efficiency.
* Repeat the cycle for each new feature.

**The TDD Cycle: Red-Green-Refactor**

This cycle is the heartbeat of TDD:

**1. Red**

Write a failing test that defines the desired behavior. This ensures the test is meaningful and the feature doesn’t yet exist.

**2. Green**

Write the simplest code to make the test pass. Focus on functionality, not optimization.

**3. Refactor**

Improve the code’s structure without changing its behavior. Clean up duplication, improve naming, and enhance readability.

This cycle is repeated for every new feature or change, keeping the codebase clean and well-tested.

**TDD Example: Function to Check if a Number Is Prime**

Let’s build a is\_prime(n) function using TDD in Python.

**Step 1: Write a failing test**

def test\_is\_prime():

assert is\_prime(5) == True

assert is\_prime(4) == False

**Step 2: Write minimal code to pass the test**

def is\_prime(n):

if n < 2:

return False

for i in range(2, int(n\*\*0.5) + 1):

if n % i == 0:

return False

return True

**Step 3: Refactor**

* Add more test cases (e.g., is\_prime(0), is\_prime(1))
* Optimize loop if needed
* Ensure readability and maintainability

**TDD vs Traditional Testing**

|  |  |  |
| --- | --- | --- |
| **Feature** | **TDD** | **Traditional Testing** |
| Timing of Tests | Before writing code | After writing code |
| Focus | Specification and design | Validation and bug detection |
| Code Coverage | Typically high | Varies |
| Feedback Loop | Immediate | Delayed |
| Design Influence | Strong | Weak |
| Learning Curve | Steeper | Easier for beginners |

**Summary:**

* **TDD** promotes clean design, early bug detection, and high test coverage.
* **Traditional testing** is better suited for exploratory testing and large-scale validation.