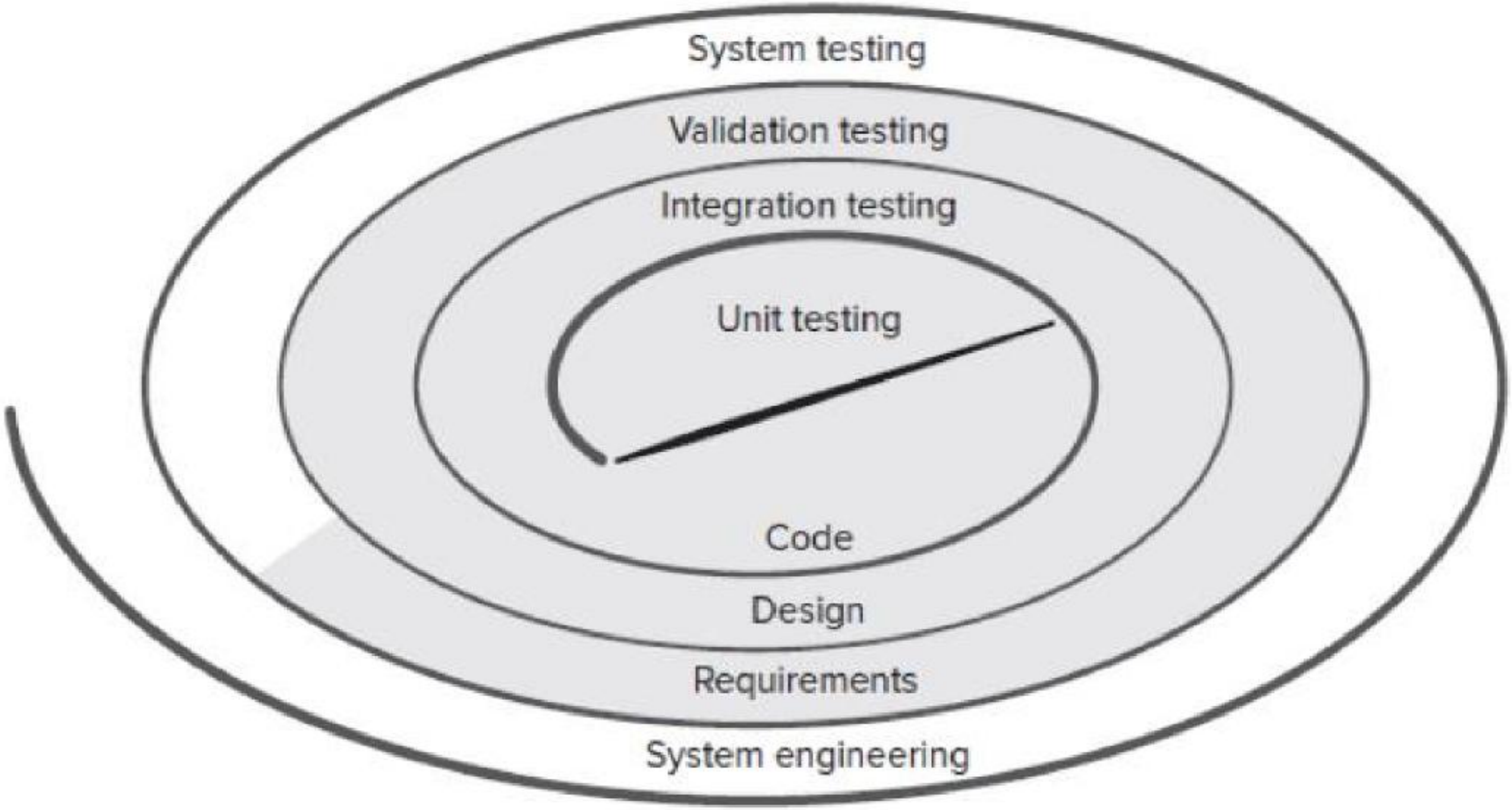
Software Testing Strategy Diagram



Software Testing Strategy Diagram

- The diagram represents a **layered software testing strategy**.
  It shows that testing progresses from the smallest software units to the complete operational system. Each outer layer covers a broader scope than the inner one.

Software Testing Strategy Diagram

1- Unit Testing

- This is the innermost layer.
- A unit is the smallest testable component of the software, such as a function, method, or class.
- Unit testing focuses on verifying that each unit behaves correctly in isolation, independent of other components.
- Its main objectives are to detect defects early, validate the correctness of the implemented logic, and reduce the cost of later rework.

Software Testing Strategy Diagram

2- Integration Testing

- Once individual units have been verified, they are combined, and their interactions are tested.
- Integration testing examines how modules exchange data and cooperate to provide higher-level functionality.
- The goal is to reveal defects that arise from incorrect interfaces, data mismatches, or misunderstandings between components.

Software Testing Strategy Diagram

4- Validation Testing

- At this level, the focus shifts from internal structure to **user and stakeholder requirements**.
- The central question becomes: Does the implemented system satisfy the specified requirements and user expectations?
- Validation testing often uses scenarios, use cases, and acceptance criteria derived directly from the requirements document.

Software Testing Strategy Diagram

5- System Testing

- This is the outermost layer, where the **entire system** is tested in an environment that is as close as possible to the real operational environment.
- System testing includes performance, security, reliability, usability, compatibility, and recovery tests.
- Its purpose is to ensure that the system operates as a coherent whole and is ready for deployment.

Software Testing Strategy Diagram

- The lower part of the diagram (Code → Design → Requirements → System Engineering) indicates that these testing activities are closely related to the development artifacts: unit tests relate to code, integration tests to design and component interactions, validation tests to requirements, and system tests to the overall engineered system.

Applying the Strategy to a Restaurant Application

Consider a **Restaurant Application** that supports online ordering, kitchen operations, delivery tracking, and management reporting.

Applying the Strategy to a Restaurant Application

1- Unit Testing in the Restaurant App

Example units: price calculation function, discount calculator, login validator, “add item to cart” method.

Unit tests verify, for instance, that **calculatorTotal ()** correctly computes the order amount, or that the login validator rejects invalid credentials.

Applying the Strategy to a Restaurant Application

- **Integration Testing in the Restaurant App**
- Typical integrations:
  - Customer interface ↔ Cart service
  - Cart service ↔ Payment gateway
  - Order service ↔ Kitchen display system
  - Order service ↔ Driver assignment module
- Integration testing verifies that an order created by the customer is correctly passed through these services, with correct data formats and states.

Applying the Strategy to a Restaurant Application

- **Validation Testing in the Restaurant App**
- Here we ask: Does the system fulfil what restaurant owners and customers requested in the requirements?
- Example checks:
    - Can a customer place and cancel an order as specified?
    - Are payment options (cash, card, wallet) supported as required?
    - Do restaurant staff receive clear and timely order information?
- These tests are usually expressed as acceptance scenarios derived from the requirements specification.

Applying the Strategy to a Restaurant Application

5- System Testing in the Restaurant App

1. The entire workflow is exercised end-to-end, e.g.:

Customer registers, browses the menu, places an order, pays, the kitchen prepares it, a driver is assigned, the order is delivered, and management reports are updated.

2. Non-functional aspects are also tested: performance under heavy load, security of customer data, resilience to failures, and usability of the interfaces.