Unit Integration System Testing

differentiate between unit, integration and system testing

Differences Between Unit Testing, Integration Testing, and System Testing

Aspect	Unit Testing	Integration Testing	System Testing
Definition	Testing individual units or components of code in isolation.	Testing the interaction between integrated units or modules.	Testing the complete and fully integrated system.
Purpose	Verify that each unit works as expected.	Ensure modules or components work together correctly.	Validate that the entire system meets the specified requirements.
Level	Lowest level of testing.	Intermediate level of testing.	High-level testing.
Focus	Functionality of a specific function, method, or module.	Data flow and interaction between modules.	End-to-end functionality and user scenarios.
Performed By	Developers (often automated).	Developers or testers.	QA/Testers.
Tools	JUnit, NUnit, PyTest.	Selenium, Postman, JMeter (API Testing).	Selenium, JMeter, TestComplete.
Test Environment	Controlled environment, often with mocked dependencies.	Partial system integration, stubs, and drivers used.	Real or production-like environment.
Input	Specific inputs for the unit under test.	Outputs from individual modules as inputs for others.	Real-world scenarios and workflows.
Example	Testing a single login function.	Testing login function with a database connection.	Testing the entire login, dashboard, and logout workflow.

Key Differences Summarized:

1. **Scope:** Unit testing focuses on individual pieces, integration testing focuses on module interaction, and system testing validates the entire system.
2. **Dependencies:** Unit tests isolate components, integration tests check interactions, and system tests involve real-world use cases.
3. **Timing:** Unit testing happens early in development, integration follows unit testing, and system testing happens before release.

Each type of testing plays a crucial role in ensuring software quality at different stages of development.

desgin test cases using one testing based on :
i. Single Fault Theory
ii. Multiple Fault Theory

Test Case Design Based on Single Fault Theory and Multiple Fault Theory

1. Single Fault Theory

- **Definition:** Assumes that **only one fault exists at a time** in the system or software under test.
- **Goal:** Identify and isolate individual faults effectively without interference from other faults.
- **Focus:** Testing each fault independently.

Example Scenario: Testing a **Login Functionality** on a website.

Test Case ID	Test Scenario	Input Data	Expected Output	Remarks
SFT-01	Valid Login Credentials	Username: user1, Password: pass123	Successful Login	Single fault: Invalid input avoided
SFT-02	Invalid Username	Username: invalid, Password: pass123	Error: Invalid Username	Fault isolated to username
SFT-03	Invalid Password	Username: user1, Password: wrongpass	Error: Invalid Password	Fault isolated to password
SFT-04	Empty Username Field	Username: [blank], Password: pass123	Error: Username Required	Fault isolated to empty username
SFT-05	Empty Password Field	Username: user1, Password: [blank]	Error: Password Required	Fault isolated to empty password

Key Assumption: Each test case is run independently, ensuring only one fault is tested at a time.

---

2. Multiple Fault Theory

- **Definition:** Assumes that **multiple faults may exist simultaneously** in the system or software under test.
- **Goal:** Identify combinations of faults and their collective impact on the system.
- **Focus:** Testing multiple fault scenarios together.

Example Scenario: Testing a **Login Functionality** on a website.

Test Case ID	Test Scenario	Input Data	Expected Output	Remarks
MFT-01	Invalid Username & Invalid Password	Username: invalid, Password: wrongpass	Error: Invalid Username/Password	Combined fault scenario
MFT-02	Empty Username & Empty Password	Username: [blank], Password: [blank]	Error: Username & Password Required	Both fields are empty
MFT-03	Valid Username & Empty Password	Username: user1, Password: [blank]	Error: Password Required	Cross-validation fault

Test Case ID	Test Scenario	Input Data	Expected Output	Remarks
MFT-04	SQL Injection in Username & Password	Username: ' OR '1'='1, Password: ' OR '1'='1	Error: SQL Injection Detected	Security fault testing
MFT-05	Valid Username & Invalid Password	Username: user1, Password: wrongpass	Error: Invalid Password	Multiple inputs tested together

Key Assumption: Test cases simulate combinations of faults to observe system behavior under complex conditions.

Comparison Between the Two Theories:

Aspect	Single Fault Theory	Multiple Fault Theory
Fault Scope	Tests one fault at a time.	Tests multiple faults together.
Complexity	Easier to design and debug.	Complex to design and debug.
Fault Detection	Good for isolated faults.	Good for fault interactions.
Testing Focus	Unit-level faults.	System-level fault interactions.
Example Use Case	Login credentials validation.	Security and integration testing.

Both approaches are essential and complement each other. **Single Fault Theory** is effective during **unit testing**, while **Multiple Fault Theory** is better suited for **integration and system testing**.

Explain V lifecycle model.

V-Model (Verification and Validation Model)

The **V-Model**, also known as the **Verification and Validation Model**, is a **software development lifecycle (SDLC)** model that emphasizes a **sequential process** where **each development phase is directly associated with a corresponding testing phase**.

The "V" shape reflects the relationship between **development phases** (on the left side of the "V") and **testing phases** (on the right side of the "V").

Key Phases of the V-Model

1. Verification Phase (Left Side of the V)

Verification ensures that the **product is being built correctly** according to requirements and design specifications.

Phase	Description	Deliverable
Requirements Analysis	Understand and document user requirements.	Requirement Specification Document

Phase	Description	Deliverable
System Design	Design the architecture of the system.	System Design Document
High-Level Design (HLD)	Define modules, interfaces, and data flow.	High-Level Design Document
Low-Level Design (LLD)	Detailed design of individual modules.	Low-Level Design Document
Coding/Implementation	Write code based on LLD.	Source Code

2. Validation Phase (Right Side of the V)

Validation ensures that the **product meets user requirements** and works as intended.

Phase	Description	Deliverable
Unit Testing	Test individual components or modules.	Unit Test Cases & Results
Integration Testing	Verify the interaction between integrated modules.	Integration Test Results
System Testing	Test the entire system for functionality and performance.	System Test Results
User Acceptance Testing (UAT)	Validate the system against user requirements.	UAT Results & Sign-off

3. Maintenance Phase (After Validation)

- After deployment, the software enters the **maintenance phase** where bug fixes, updates, and enhancements are performed.

Diagram of V-Model

```sql
Requirements Analysis      →       User Acceptance Testing
       ↑                                    ↓
System Design              →       System Testing
       ↑                                    ↓
High-Level Design (HLD)    →       Integration Testing
       ↑                                    ↓
Low-Level Design (LLD)     →       Unit Testing
       ↑                                    ↓
Coding/Implementation
```

Advantages of V-Model

1. **Clear Structure:** Each phase has clear deliverables and associated test plans.
2. **Early Detection of Defects:** Testing is planned in parallel with development, reducing defect costs.
3. **Better Documentation:** Each phase is well-documented and structured.
4. **Easy to Manage:** Well-defined phases make project tracking straightforward.

Disadvantages of V-Model

3

1. **Rigid and Inflexible:** Changes in requirements during development are hard to accommodate.
2. **Expensive for Small Projects:** Overhead from documentation and verification/testing phases.
3. **Limited Flexibility:** Not suitable for projects with frequently changing requirements.

When to Use V-Model?

- When requirements are **clear and well-defined**.
- For **small to medium-sized projects** with minimal changes expected.
- In projects where **high reliability and quality assurance** are critical (e.g., healthcare, aerospace, banking).

The **V-Model** ensures that **every development stage has a corresponding testing stage**, making it highly effective for projects requiring high quality and reliability.

Explain verfication and validation. What are differences between the two?

Verification vs Validation in Software Testing

Verification and **Validation** are two critical components of software testing, ensuring that software meets **technical specifications** and **user requirements**, respectively.

---

1. Verification

- **Definition:** Verification is the process of **evaluating work products** (like documents, design, code) at various stages of the software development lifecycle (SDLC) to ensure they meet **specified requirements**.
- **Focus:** Are we **building the product correctly**?
- **Objective:** Ensure the software matches the **design specifications**.
- **When it Happens:** During the **development phase** of the SDLC.
- **Activities Involved:**
  - Reviews (Code Reviews, Design Reviews)
  - Walkthroughs
  - Inspections
- **Example:** Checking if the login page was developed as per the design document.

---

2. Validation

- **Definition:** Validation is the process of **evaluating the final product** to ensure it meets the **business and user requirements**.
- **Focus:** Are we **building the right product**?
- **Objective:** Ensure the **end product satisfies user needs and expectations**.
- **When it Happens:** After the development phase, during **testing and deployment**.
- **Activities Involved:**

  - Unit Testing
  - Integration Testing
  - System Testing
  - User Acceptance Testing (UAT)
- **Example:** Checking if the login page allows users to successfully log in with valid credentials.

---

3. Key Differences Between Verification and Validation

Aspect	Verification	Validation
Definition	Ensures the product is built **correctly** as per specifications.	Ensures the product is built to meet **user requirements**.
Focus	Design, architecture, and code.	Final product functionality and usability.
Objective	Ensure compliance with **technical requirements**.	Ensure compliance with **business needs**.
Phase	Happens during **development** phases.	Happens after **development**, during **testing**.
Activities	Reviews, walkthroughs, inspections.	Testing (Unit, Integration, System, UAT).
Performed By	Developers, technical leads.	Testers, QA teams, end users.
Questions Answered	Are we building the product **correctly**?	Are we building the **right product**?
Example	Checking if the database schema meets design specs.	Checking if the app performs as expected in real scenarios.

---

4. Real-World Analogy

- **Verification:** Checking the blueprint of a building before starting construction to ensure everything is planned correctly.
- **Validation**: Inspecting the completed building to ensure it is safe, functional, and meets the intended purpose.

Both **Verification** and **Validation** are equally important for delivering **high-quality software** that meets both **technical requirements** and **user expectations**.

Differentiate between software quality assurance and software testing

Difference Between Software Quality Assurance (SQA) and Software Testing

Aspect	Software Quality Assurance (SQA)	Software Testing
Definition	A **proactive process** focused on ensuring software quality **throughout the development lifecycle**.	A **reactive process** focused on **identifying defects** in the software after development.

Aspect	Software Quality Assurance (SQA)	Software Testing
Focus	Preventing defects and ensuring quality **processes** are followed.	Detecting and fixing **defects** in the software product.
Objective	To **improve and optimize processes** to avoid defects.	To **identify bugs and errors** in the software product.
Approach	**Process-oriented:** Ensures standards, guidelines, and frameworks are followed.	**Product-oriented:** Validates functionality, performance, and correctness of the product.
When it Happens	**Throughout the software development lifecycle (SDLC)**.	**After the development phase**, during testing stages.
Techniques Used	Audits, Reviews, Walkthroughs, Process Improvement.	Unit Testing, Integration Testing, System Testing, UAT.
Responsibility	SQA Team or QA Engineers.	Testers or QA Testers.
Tools Used	JIRA, Confluence, Bugzilla (for process management).	Selenium, JUnit, TestNG (for software testing).
Outcome	Ensures **prevention** of defects through well-defined processes.	Ensures **detection** of defects before release.
Example	Ensuring the team follows coding standards and best practices.	Running test cases to check if the login functionality works correctly.

Key Differences Summarized:

1. **Proactive vs Reactive:**
    - **SQA:** Prevent defects by improving processes.
    - **Testing:** Detect defects in the finished product.
2. **Scope:**
    - **SQA:** Focuses on the **entire SDLC** process.
    - **Testing:** Focuses on **specific phases** of SDLC, primarily after development.
3. **Outcome:**
    - **SQA:** Quality assurance of the **development process**.
    - **Testing:** Quality assurance of the **final product**.

Analogy:

- **SQA:** Like ensuring chefs follow the correct recipe and kitchen hygiene practices while cooking.
- **Testing:** Like tasting the dish to ensure it tastes as expected before serving it to the customer.

Both **SQA** and **Software Testing** are complementary processes. **SQA** sets the stage for quality, while **Testing** validates that the quality goals have been achieved.

Differentiate between any two software testing tools.

Difference Between Selenium and JUnit

Aspect	Selenium	JUnit
Type of Tool	Automation Testing Framework	Unit Testing Framework
Purpose	Used for **automated testing** of **web applications**.	Used for **unit testing** of **Java applications**.
Scope	Focuses on **UI testing**, cross-browser testing, and regression testing.	Focuses on **unit-level testing** of individual methods and classes.
Language Support	Supports multiple languages like **Java, Python, C#, JavaScript, Ruby**.	Primarily supports **Java**.
Test Environment	Tests applications through **web browsers**.	Tests applications in an **isolated environment**.
Test Type	Functional, regression, UI testing.	Unit testing.
Execution Style	Executes tests across **multiple browsers** and platforms.	Executes tests within the **Java runtime environment**.
Integration	Integrates with **TestNG, Jenkins, Maven** for CI/CD.	Often used with **Maven, Jenkins** for build automation and CI/CD.
Learning Curve	Steeper learning curve, requires knowledge of **browser automation**.	Easier to learn for **Java developers**.
Example Use Case	Automating the testing of a login form on a website across browsers.	Testing a method in a Java class to ensure it returns the expected output.

Key Differences Summarized:

1. **Testing Scope:** Selenium is for **web application UI testing**, while JUnit is for **unit testing Java code**.
2. **Language Support:** Selenium supports **multiple programming languages**, while JUnit is **Java-specific**.
3. **Testing Focus:** Selenium is **browser-based**, whereas JUnit works in a **Java runtime environment**.

Analogy:

- **Selenium:** Like testing if the **entire car's dashboard functions correctly**.
- **JUnit:** Like checking if the **engine's spark plug works properly**.

If you're testing **web application functionality across browsers**, use **Selenium**. If you're testing **individual Java methods**, use **JUnit**.

Consider program for finding a number is prime or not. Write test cases using one white box testing and one black box testing.

Program Example: Check if a Number is Prime

python

```python
def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True
```

---

1. White Box Testing Test Cases

Technique Used: *Statement Coverage*

Objective: Ensure all code paths and logic branches are tested.

Test Case ID	Input	Expected Output	Reasoning
TC_WB_01	`-1`	`False`	Covers the condition `if num <= 1` (Negative number).
TC_WB_02	`0`	`False`	Covers the condition `if num <= 1` (Zero).
TC_WB_03	`1`	`False`	Covers the condition `if num <= 1` (One is not prime).
TC_WB_04	`2`	`True`	Covers the loop condition where the range doesn't execute (`for` loop skipped).
TC_WB_05	`4`	`False`	Tests the loop where a divisor is found (`4 % 2 == 0`).
TC_WB_06	`7`	`True`	Tests a number that passes through the loop without finding a divisor.

---

2. Black Box Testing Test Cases

Technique Used: *Equivalence Partitioning and Boundary Value Analysis*

Objective: Test valid and invalid input partitions and boundary values.

Test Case ID	Input	Expected Output	Test Type	Reasoning
TC_BB_01	`-5`	`False`	Invalid Input	Negative numbers are not prime.
TC_BB_02	`0`	`False`	Boundary Value	Zero is not prime.
TC_BB_03	`1`	`False`	Boundary Value	One is not prime.
TC_BB_04	`2`	`True`	Valid Input	2 is the smallest prime number.
TC_BB_05	`9`	`False`	Valid Input	9 is divisible by 3.
TC_BB_06	`13`	`True`	Valid Input	13 is a prime number.

Summary:

- **White Box Testing:** Focused on testing **code logic and branches**.
- **Black Box Testing:** Focused on testing **input-output behavior** without looking into internal code logic.

These test cases ensure that the program is thoroughly validated for **both logical correctness and expected outputs** across different testing paradigms.

What is cyclomatic and code complexity ?
What are the differences between them?

Cyclomatic Complexity vs Code Complexity

While both terms deal with evaluating the **complexity** of software code, they have distinct meanings and focus areas.

---

1. Cyclomatic Complexity

- **Definition:** Cyclomatic complexity is a **quantitative measure of the number of independent paths** through a program's source code.
- **Purpose:** It helps to **identify the complexity of control flow** in a program.
- **Focus:** Focuses on the **decision points** in the code (e.g., `` `if` ``, `` `else` ``, `` `for` ``, `` `while` ``, `` `switch` ``).
- **Measurement Metric:** Calculated using the formula:

$$M = E - N + 2P$$

Where:

- **M**: Cyclomatic Complexity
- **E**: Number of edges in the control flow graph
- **N**: Number of nodes in the control flow graph
- **P**: Number of connected components (usually `` `1` `` for a single program)
- **Ideal Value:** A cyclomatic complexity score of **10 or lower** is generally considered maintainable.
- **Example:**

```python
if (x > 0):
    print("Positive")
else:
    print("Negative")
```

  - Cyclomatic Complexity = **2** (Two independent paths).

---

2. Code Complexity

- **Definition:** Code complexity is a **broader term** that refers to the **overall difficulty of understanding, maintaining, and modifying a piece of code**.
- **Purpose:** Evaluates **readability, maintainability, and logical structure** of the code.
- **Focus:** Considers various factors:

  - Code length
  - Nested loops
  - Depth of conditional statements
  - Use of recursion
  - Dependency between modules
- **Measurement Metric:** There's **no single formula**. Metrics like **Halstead Complexity Measures**, **Lines of Code (LOC)**, and **Maintainability Index** are used.
- **Ideal Value:** Lower values generally indicate **simpler** and **more maintainable code**.
- **Example:** A deeply nested loop with multiple conditionals can indicate high code complexity even if cyclomatic complexity is low.

---

3. Differences Between Cyclomatic Complexity and Code Complexity

Aspect	Cyclomatic Complexity	Code Complexity
Definition	Measures the number of **independent paths** in the code.	Measures the **overall complexity** of code, including readability and structure.
Focus	Focuses on **control flow** and decision points.	Considers **logic, structure, length, and readability**.
Metric/Formula	Based on the formula **M = E - N + 2P**.	No single formula; metrics like **LOC, Halstead Measures, Maintainability Index** are used.
Scope	Limited to **control structures** (`if`, `else`, `for`, `while`).	Encompasses the **entire codebase**, including code length, dependencies, and nesting.
Ideal Value	A value **≤ 10** is generally considered manageable.	Varies based on the metric being used.
Measurement Tool	Tools: **SonarQube, CodeClimate**.	Tools: **SonarQube, PMD, Checkstyle**.
Use Case	Useful for **unit testing and branch testing**.	Useful for **code maintainability and refactoring**.

---

4. Real-World Analogy

- **Cyclomatic Complexity:** Counting the number of **routes in a traffic intersection** to estimate how many different paths cars can take.
- **Code Complexity:** Evaluating **how congested and well-organized the entire traffic system is**, including road quality, traffic lights, and pedestrian crossings.

Both metrics are **complementary** — Cyclomatic Complexity helps optimize **individual functions or methods**, while Code Complexity provides a **holistic view of the entire codebase**.

Consider any simple program and generate all possible slices for that program. Design atleast one test case for each slice.

Program Example: Calculate the Sum and Product of Two Numbers

python

```python
def calculate_sum_and_product(a, b):
    sum_result = a + b           # Slice 1
    product_result = a * b       # Slice 2
    if sum_result > product_result:  # Slice 3
        return "Sum is greater"
    else:                        # Slice 4
        return "Product is greater or equal"
```

---

1. Program Slices and Test Cases

Slice 1: `sum_result = a + b`

- **Focus:** Test the addition operation independently.
- **Test Case:**
  - **Input:** `a = 2, b = 3`
  - **Expected Output:** `sum_result = 5`

Slice 2: `product_result = a * b`

- **Focus:** Test the multiplication operation independently.
- **Test Case:**
  - **Input:** `a = 2, b = 3`
  - **Expected Output:** `product_result = 6`

Slice 3: `if sum_result > product_result:`

- **Focus:** Test the condition where the sum is greater than the product.
- **Test Case:**
  - **Input:** `a = 1, b = 1`
  - **sum_result = 2**, **product_result = 1**
  - **Expected Output:** `"Sum is greater"`

Slice 4: `else:`

- **Focus:** Test the condition where the product is greater than or equal to the sum.
- **Test Case:**
  - **Input:** `a = 2, b = 3`
  - **sum_result = 5**, **product_result = 6**
  - **Expected Output:** `"Product is greater or equal"`

---

2. Explanation of Slices and Test Case Design

1. **Slice 1 (Addition Operation):** Focuses on testing the **correctness of sum calculation** without considering other program paths.
2. **Slice 2 (Multiplication Operation):** Focuses on testing **multiplication independently**.

3. **Slice 3 (Conditional - Sum > Product):** Tests the **'if' condition** logic.
4. **Slice 4 (Conditional - Else Block):** Tests the **'else' block** logic.

---

3. Complete Test Case Table

Slice ID	Test Objective	Input (a, b)	Expected Output
S1	Validate sum operation	`2, 3`	`sum_result = 5`
S2	Validate product operation	`2, 3`	`product_result = 6`
S3	Validate if condition	`1, 1`	`"Sum is greater"`
S4	Validate else condition	`2, 3`	`"Product is greater or equal"`

---

4. Why Program Slices are Useful in Testing?

- **Isolation of Logic:** Each slice tests one logical part of the code.
- **Targeted Testing:** Helps focus on **individual behaviors** without interference from other logic paths.
- **Debugging:** Easier to identify bugs in isolated slices.

This approach ensures comprehensive testing of each independent path in the program.