UNIT V – TEST-DRIVEN DEVELOPMENT

Object-Oriented Testing: Object-Oriented Unit Testing, Object-Oriented Integration Testing, Object-Oriented System Testing - Software Complexity: Unit-Level Complexity, Integration-Level Complexity, System- Level Complexity - Model-Based Testing for Systems of Systems: Characteristics, Sample Systems of Systems Software Engineering for Systems of Systems.

Software Complexity

Software Complexity refers to the level of difficulty involved in understanding, developing, testing, maintaining, and modifying a software system. As a software system becomes larger and contains more classes, methods, data, and interactions, its complexity also increases. In Object-Oriented Testing, software complexity is mainly considered at three levels: Unit-Level Complexity, Integration-Level Complexity, and System-Level Complexity. Unit-Level Complexity focuses on the complexity of an individual class or program unit. Integration-Level Complexity deals with the complexity created when different classes or units interact with each other. System-Level Complexity considers the complexity of the complete software system and the interactions among its different components. Studying software complexity helps testers identify difficult areas and design suitable test cases.

1. Unit-Level Complexity

Unit-Level Complexity refers to the complexity of a single program unit, such as a class, method, or function. It focuses on the internal logic of that individual unit, including the number of conditions, loops, variables, and operations used. A unit with more decision-making and complex logic is more difficult to understand and test.

Example: In an Online Shopping System, consider a Product class. If the class contains methods such as addProduct(), updatePrice(), checkStock(), and calculateDiscount(), the complexity of these methods contributes to the Unit-Level Complexity.

2. Integration-Level Complexity

Integration-Level Complexity refers to the complexity created when two or more program units, classes, or components interact with each other. It mainly focuses on how data and messages are exchanged between different units. As the number of interactions increases, testing the connections between the units becomes more difficult.

Example: In an Online Shopping System, the Product, ShoppingCart, Payment, and Order classes may interact with one another. The ShoppingCart gets product details from the Product class, sends the total amount to the Payment class, and passes order information to the Order class. These interactions create Integration-Level Complexity.

3. System-Level Complexity

System-Level Complexity refers to the complexity of the complete software system and the relationships among its major components. It considers the overall behavior of the system,

including different modules, workflows, external services, users, and their interactions. Testing becomes more complex because many components must work together correctly.

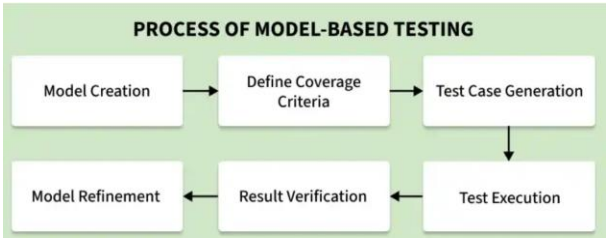
Example: In an Online Shopping System, the complete process may involve user login, product search, product selection, shopping cart, order placement, payment, inventory update, and delivery tracking. All these components must work together properly. The complexity of the complete Online Shopping System is called System-Level Complexity.

Model-Based Testing for Systems of Systems:
Model-Based Testing (MBT) is a software testing technique in which test cases are created from models that describe how a system behaves. The model represents the expected behavior, states, actions, and interactions of the system. Based on this model, test cases can be generated automatically or with limited manual involvement. MBT helps testers achieve better test coverage, maintain consistency in testing, and reduce the amount of manual work.

Uses Models: Models such as **Finite State Machines (FSM), UML diagrams, and Statecharts** can be used to represent system behavior.
Test Case Generation: Test cases can be generated **automatically or semi-automatically** from the model.
Better Testing: It improves **test coverage and consistency** while reducing manual testing effort.

Process of Model-Based Testing

Model-Based Testing (MBT) follows a structured process where a system is modeled, test cases are generated from the model, executed on the system, and results are analyzed to improve quality.



Steps in Model-Based Testing

1. **Model Creation:**
   The requirements, user stories, or expected system behavior are represented using a formal model such as **FSM, EFSM, or UML diagrams**. The model describes how the system is expected to work.
2. **Define Coverage Criteria:**
   The testing goals are decided to determine how much of the model should be tested. Common criteria include **state coverage, transition coverage, and path coverage**.

3. **Test Case Generation:**
   Test cases are created from the model using suitable algorithms or model-traversal methods. This process can be done **automatically or semi-automatically**.
4. **Test Execution:**
   The generated test cases are run on the actual software using testing tools or automation frameworks.
5. **Result Verification:**
   The actual output of the software is compared with the expected output obtained from the model. Differences between them may indicate defects in the system.
6. **Model Refinement:**
   When errors or differences are identified, the model is modified or improved. The updated model can then be used for better testing in the next testing cycle.

Types of Model-Based Testing

Model-Based Testing uses different kinds of models to describe the **behavior, rules, and interactions** of a software system. These models provide a systematic way to create test cases and help achieve better test coverage.

- **Finite State Machines (FSM) / State Transition Models:**
  These models describe a system using different **states and transitions**. A transition occurs when a particular event or input is received. They are useful for testing workflows and behavior that depends on the current state.
- **Statecharts:**
  Statecharts are an extended version of FSMs. They can represent **hierarchical, parallel, and complex states**. They are commonly used to test reactive and event-driven systems.
- **Decision Tables:**
  Decision tables represent **business rules using rows and columns**. They show how different combinations of conditions produce specific actions or results. They are useful for testing systems with complex business rules.
- **UML-Based Models:**
  UML models use diagrams such as **Use Case, Sequence, Activity, and State diagrams** to represent system behavior and design. Test cases can be prepared from the requirements and design represented in these diagrams.
- **Markov Models:**
  Markov models represent a system using **probability-based transitions between states**. They are mainly used for testing areas such as **reliability, performance, and actual usage patterns**.
- **Formal Specification Models:**
  These models use **mathematical notations and precise specifications** to describe system behavior. They are mainly useful for **safety-critical systems**, where correctness and reliability are very important.

Characteristics of Systems of Systems (SoS)

Systems of Systems (SoS) refers to a collection of **independent systems that work together** to achieve a common goal. Each individual system can operate independently, but when they are connected, they provide additional overall functionality

Operational Independence
Each system in the SoS can perform its own functions independently. It can continue to operate even when it is not connected to the other systems.

Managerial Independence
Each individual system has its own management, control, and objectives. The systems may be managed by different organizations or authorities.

Geographical Distribution
The systems may be located in different places or geographical areas. They communicate and work together through communication networks.

Emergence
When independent systems work together, they can provide new capabilities or behaviors that are not available from any single system alone.

Evolutionary Development
An SoS can change and develop over time. Individual systems may be upgraded, replaced, or added without completely rebuilding the entire system.

Sample Systems of Systems (SoS)

A **System of Systems (SoS)** is formed when multiple independent systems work together to achieve a common objective. Each individual system can operate independently, but their interaction provides additional capabilities.

Examples include a smart transportation system, where traffic signals, GPS, public transport, and emergency services work together; a healthcare system, where hospitals, laboratories, ambulance services, and medical information systems interact; and a smart city system, where electricity, water supply, transportation, security, and communication systems are connected. These systems demonstrate how independent systems can cooperate to provide a larger and more useful overall service.

Software Engineering for Systems of Systems

Software Engineering for Systems of Systems (SoS) is the process of developing, integrating, testing, and maintaining software that connects multiple independent systems. Since each system may have its own requirements, technology, data, and management, SoS development requires careful planning and coordination. Engineers must focus on communication between systems, data exchange, interoperability, security, reliability, and overall system behavior.

For example, in a smart transportation system, software engineering is used to connect traffic signals, GPS systems, public transport, and emergency services so that they can exchange

information and work together effectively. Testing is also important to ensure that both individual systems and their overall interactions function correctly.