Assignment 2

Q1 Difference between Function and Services in Software Systems.

Difference Between Function and Services in Software Systems:

Aspect	Function	Service
Definition	A **function** is a block of code designed to perform a specific task within a software system. It typically takes input, processes it, and returns output.	A **service** is a broader, self-contained module that provides reusable, scalable functionality accessible over a network or within a system.
Scope	Functions are **local** to the application or program they belong to.	Services are designed to be **independent** and can be accessed across different applications or systems.
Purpose	Functions focus on performing a specific **operation** or computation.	Services focus on providing **features** or capabilities to users or other software.
Reusability	Functions are often reused within the same application.	Services are **highly reusable** and can be consumed by various applications, often through APIs.
Interaction	Functions are **called directly** in the same application or codebase.	Services are typically accessed via **APIs**, web services, or microservices architecture.
Real-World Example	A function like calculateTax() in an e-commerce app to compute tax based on product price.	A payment processing **service** like PayPal or Stripe, which is integrated into multiple e-commerce platforms.
Deployment	Functions are deployed within the application's environment.	Services can be deployed **separately** and managed independently, such as a cloud service.
State	Functions are typically **stateless** and do not maintain state between calls.	Services may be **stateful** or stateless, depending on the design.
Communication	Functions communicate within the same application (via function calls).	Services communicate over a network using **protocols** (e.g., HTTP, SOAP, or REST).

Example from Software Development	A function in Python like def add_numbers(a, b): return a + b.	A RESTful API service providing weather updates like **OpenWeatherMap**.

This distinction helps clarify how functions are the smaller building blocks of code, while services represent larger, independent components that offer functionality across systems.

Q2 Make Use Case Diagram/Class Diagram.

1. Use Case Diagram:

- A Use Case Diagram is a visual representation of the interactions between users (actors) and the system. It illustrates the different functionalities (use cases) the system provides and how various actors (such as users, administrators, or external systems) engage with those functionalities.
- By modeling the user interactions, the diagram helps clarify system requirements and design, showing how users perform tasks like placing orders, making payments, or managing products. This diagram provides a high-level overview of the system's behavior from an external perspective, making it valuable for stakeholders to understand the system's scope and flow of operations.

Step 1: Identify Actors

Determine who or what interacts with the system. These are your actors. They can be users, other systems, or external entities.

Step 2: Identify Use Cases

Identify the main functionalities or actions the system must perform. These are your use cases. Each use case should represent a specific piece of functionality.

Step 3: Connect Actors and Use Cases

Draw lines (associations) between actors and the use cases they are involved in. This represents the interactions between actors and the system.

Step 4: Add System Boundary

Draw a box around the actors and use cases to represent the system boundary. This defines the scope of your system.

Step 5: Define Relationships

If certain use cases are related or if one use case is an extension of another, you can indicate these relationships with appropriate notations.

Step 6: Review and Refine

Step back and review your diagram. Ensure that it accurately represents the interactions and relationships in your system. Refine as needed.

Step 7: Validate

Share your use case diagram with stakeholders and gather feedback. Ensure that it aligns with their understanding of the system's functionality.

Online Shopping System:

1. Actors:

- Customer
- Admin

2. Use Cases:

1. Browse Products
2. Add to Cart
3. Checkout
4. Manage Inventory (Admin)

3. Relations:

- The Customer can browse products, add to the cart, and complete the checkout.
- The Admin can manage the inventory.

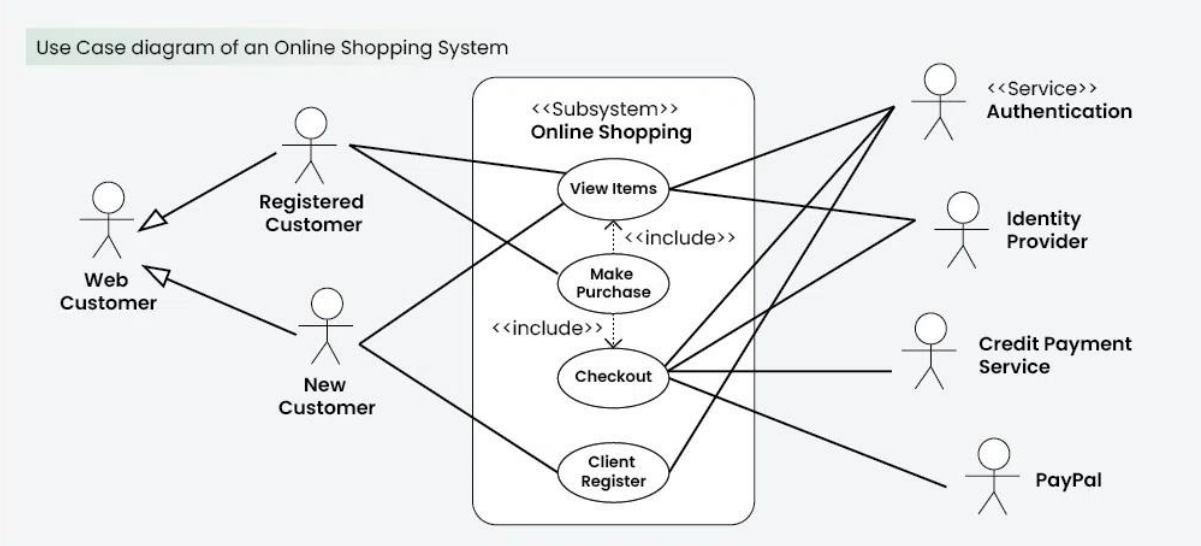

Fig: Use Case Diagram of a Online Shopping System

2. Class Diagram:

A **Class Diagram** is a fundamental part of Object-Oriented design. It represents the static structure of a system by showing the system's classes, their attributes, methods, and the

relationships among objects. In the case of an **Online Shopping System**, this diagram models the key classes involved in the online shopping process and their interactions.

This online shopping class diagram shows the domain model for online shopping. This diagram will help software engineers and business analysts easily understand the diagram. The diagram links classes like user and account to show how an order is placed and then shipped.

Detailed Components of an Online Shopping Class Diagram:

1. **Classes**: The main building blocks representing entities in the system. Each class will have:
    - **Attributes**: Represent the properties of a class.
    - **Methods**: Represent the behaviors/functions of a class.
2. **Associations**: Define how classes relate to each other. Common types of relationships include:
    - **Aggregation**: A class is a collection or container for other classes (e.g., a Shopping Cart contains Products).
    - **Composition**: A stronger form of aggregation where the child class cannot exist without the parent (e.g., an Order contains Order Items).
    - **Inheritance**: A class shares behavior and attributes from a parent class (e.g., User can be a parent class for Customer and Admin).

Main Components of an Online Shopping Class Diagram:

1. **User Class**:
    - **Attributes**: userID, name, email, password
    - **Methods**: login(), logout(), register()
    - **Description**: Represents the users of the system. There can be different types of users like Customers and Admins.
2. **Account Class**:
    - **Attributes**: accountID, balance, accountType
    - **Methods**: credit(), debit()
    - **Description**: Represents the account linked with a user, used for payment purposes. A customer can have multiple payment options linked to their account.
3. **Product Class**:
    - **Attributes**: productID, productName, description, price, stock
    - **Methods**: addProduct(), removeProduct(), updateStock()
    - **Description**: Represents the products available for purchase in the online store.

4. **Order Class**:
    - **Attributes**: orderID, orderDate, status
    - **Methods**: placeOrder(), cancelOrder(), updateOrder()
    - **Description**: Represents the order placed by the user. The Order is linked to the User and contains multiple Products.
5. **Shopping Cart Class**:
    - **Attributes**: cartID, totalItems, totalPrice
    - **Methods**: addItem(), removeItem(), checkout()
    - **Description**: Represents the shopping cart where the customer adds products before placing an order. It is linked to the user.
6. **Payment Class**:
    - **Attributes**: paymentID, paymentMethod, paymentDate, amount
    - **Methods**: processPayment(), refund()
    - **Description**: Handles the payment process when an order is placed. It is linked to the order and the user's account.
7. **Shipment Class**:
    - **Attributes**: shipmentID, shipmentDate, deliveryDate, trackingNumber
    - **Methods**: shipOrder(), trackShipment()
    - **Description**: Manages the shipping details once the order is placed. It is linked to the order and has a relationship with the shipping company.
8. **Admin Class**:
    - **Attributes**: adminID, adminName, privileges
    - **Methods**: manageProduct(), manageUser(), viewReports()
    - **Description**: Admin can add, remove, or update products, manage users, and view sales reports.

Relationships:

- **User - Account**: A **User** can have multiple **Accounts** for different payment methods (one-to-many).
- **User - Order**: A **User** can place multiple **Orders** (one-to-many).
- **Order - Product**: An **Order** contains multiple **Products** (many-to-many), typically linked through an intermediary class like **OrderItem**.
- **Order - Payment**: An **Order** has one **Payment** method.
- **Order - Shipment**: An **Order** will have a related **Shipment** once it's processed.

Example of Relationships:

- A **Customer** (a type of **User**) adds **Products** to their **Shopping Cart**.
- The **Shopping Cart** belongs to the **Customer** and contains one or more **Products**.
- After placing an **Order**, the **Customer** makes a **Payment** through one of their **Accounts**.
- The **Order** is shipped, and the **Shipment** details are tracked by the system.

Diagram Structure Summary:

- **Classes**: Represent entities like User, Order, Product.
- **Attributes**: Represent fields like orderID, productName, price.
- **Methods**: Define the behaviors like addProduct(), placeOrder().
- **Relationships**: Show how entities interact, like one-to-many, many-to-many (e.g., a User can place many Orders).

This structured diagram helps software engineers and business analysts understand how users interact with the system, from browsing products to placing orders, making payments, and receiving shipments.

Online Shopping Process UML Diagram