Assignment-01

Student Name: Tanay Manish Nesari
UID: 23BCS13761
Branch: BE-CSE
Section/Group: KRG_2B
Semester: 6th
Date of Performance: 30/1/26
Subject Name: System Design
Subject Code: 23CSH-314

Q1. Explain the role of **interfaces and enums** in software design with proper examples.
(2.5 marks)

Ans) **Interface:** An interface defines a contract that a class must follow. It specifies what a class should do, but not how it should do it. Interfaces help achieve abstraction, loose coupling, and polymorphism.

Importance of Interface:

- Promotes flexibility: different classes can implement the same interface in different ways.
- Supports multiple inheritance of behavior (a class can implement multiple interfaces).
- Makes systems easier to extend, maintain, and test.

For example, in a **Payment Processing Application**, an interface `PaymentMethod` can define a method `payAmount()`, which is implemented by classes such as `CreditCardPayment`, `DebitCardPayment`, and `UPIPayment`. This allows the application to process payments through different modes without changing the core business logic, making the system easier to maintain and extend.

```java
interface Payment {
    void pay(double amount);
}

class CreditCardPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }
}

class DebitCardPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Debit Card");
    }
}

class UPIPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using UPI");
    }
}

```

Enums: An enum (enumeration) represents a fixed set of predefined constants. It ensures that a variable can only take one of the specified values, improving type safety, readability, and consistency.

Importance of Enum:

- Prevents invalid values by restricting inputs to predefined constants.
- Improves readability and maintainability of code.
- Simplifies logic that depends on fixed categories or states.

For example, in an **Order Management System**, an enum `OrderStatus` can contain values such as `PENDING`, `CONFIRMED`, `SHIPPED`, and `DELIVERED`. Using an enum ensures that the order status can only take one of these valid states, reducing bugs and making the application logic clearer and more reliable.

```
enum OrderStatus {
    PENDING,
    CONFIRMED,
    SHIPPED,
    DELIVERED
}

class Order {
    OrderStatus status;

    Order(OrderStatus status) {
        this.status = status;
    }
}

```

Q2. Discuss how **interfaces enable loose coupling** with an example.
(2.5 marks)

Ans) **Loose Coupling:** Loose coupling refers to a design approach in which different components of a system have minimal dependencies on each other. Each component interacts with others through well-defined abstractions, allowing internal implementations to change without affecting the rest of the system.

Role of Inetrfaces in Enabling loose coupling:

- Interfaces allow programs to depend on abstractions rather than concrete classes.
- The calling class interacts only with the interface, not the specific implementation.
- Implementation changes do not affect dependent classes.
- New implementations can be added easily without modifying existing code, improving flexibility and maintainability.

.

For example, In a **Notification System**, an interface `NotificationService` defines a method `sendNotification()`. This interface is implemented by classes such as `EmailNotification` and `SMSNotification`. The main application communicates only with the `NotificationService` interface. As a result, notification methods can be changed or extended without altering the core system, achieving loose coupling and better maintainability.

```
interface NotificationService {
    void sendNotification(String message);
}

class EmailNotification implements NotificationService {
    public void sendNotification(String message) {
        System.out.println("Email sent: " + message);
    }
}

class SMSNotification implements NotificationService {
    public void sendNotification(String message) {
        System.out.println("SMS sent: " + message);
    }
}

class NotificationApp {
    private NotificationService service;

    NotificationApp(NotificationService service) {
        this.service = service;
    }

    void notifyUser(String message) {
        service.sendNotification(message);
    }
}
```

Q3. Design a **High-Level Design (HLD)** for a **Payment Processing System**, showing where **interfaces** would be used

Ans) **Functional Requirements:**

- Client should be able to initiate payment requests
- Gateway creates a temporary session for the user to enter card details
- Secure PCI-DSS compliant handling of card data
- Transaction status tracking
- Out of scope (OOS): part payments, returns

Non-Functional Requirements:

- **Scalability:** ~10,000 TPS
- Consistency priority (Availability << Consistency), strong consistency for transactions
- **Latency:** < 200 ms response time for payment authorization
- **Security:** End-to-end encryption and PCI-DSS Level-1 compliance

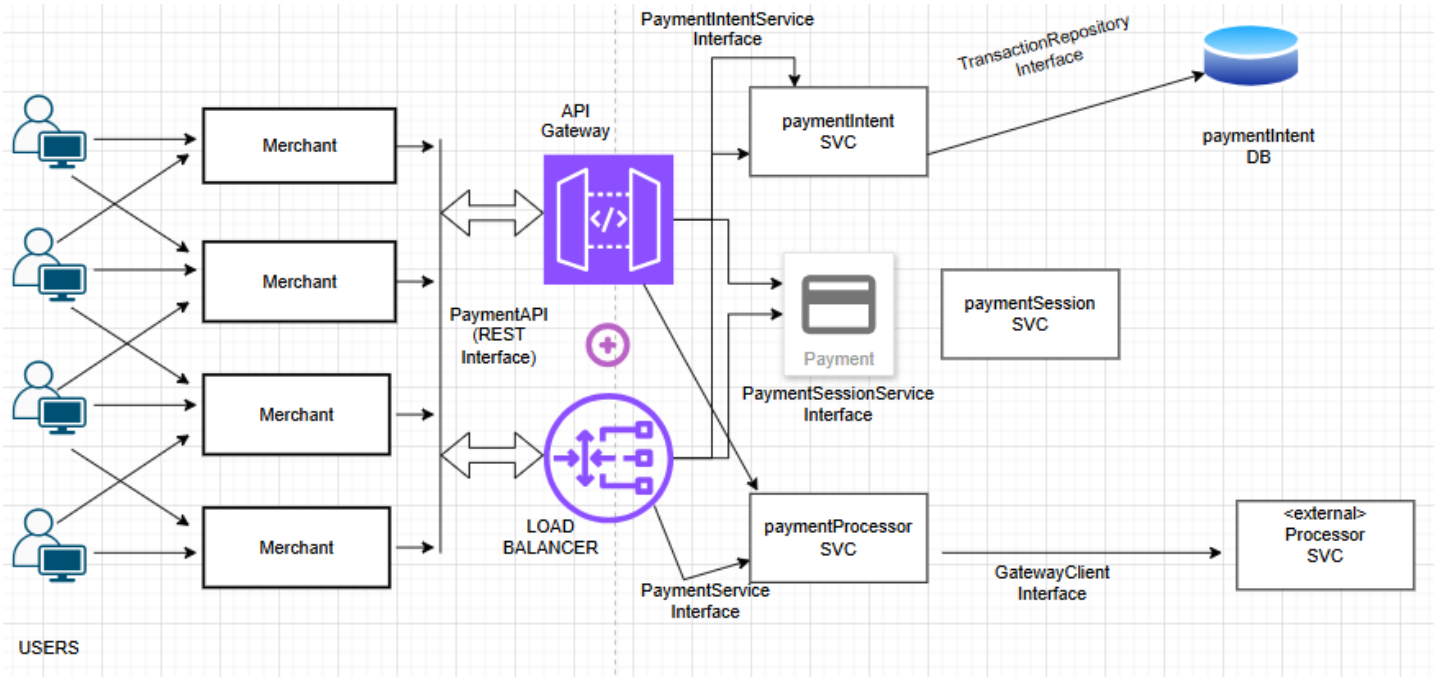
Core Entities:

- Merchant / Clients
- Transaction
- PaymentMethod
- Customer / Users
- WebhookEvent
- PaymentSession

API Designing:

- **POST /v1/payment-intents**
- **POST /v1/payment-sessions**
- **POST /v1/checkout/pay**
- **GET /v1/payments/{payment_id}**

HLD Diagram:


PaymentIntentService Interface
TransactionRepository Interface
API Gateway
paymentIntent SVC
paymentIntent DB
Merchant
Merchant
Merchant
Merchant
PaymentAPI (REST Interface)
Payment
paymentSession SVC
PaymentSessionService Interface
LOAD BALANCER
paymentProcessor SVC
<external> Processor SVC
GatewayClient Interface
PaymentService Interface
USERS