By: Muhammad Taha Nasir

Lecture Notes:

2. “Understanding the Strategy Design Pattern”

1. Introduction to Design Patterns

When building applications, developers often face recurring problems. These issues aren't unique to one project and many developers have encountered them before and found solutions. These solutions, refined over time, are called **design patterns**. They act like blueprints, guiding us on how to structure our code to solve common problems efficiently.

Design patterns help us:

- Reuse proven solutions.
- Make our code flexible and easy to update.
- Save time by avoiding trial-and-error.

Applications evolve constantly because new features get added, and requirements change. A good design ensures that adding new features requires minimal code changes and effort. This flexibility is key to creating maintainable software. Design patterns, along with principles like SOLID and Object-Oriented Programming (OOP), help us achieve this.

1.1 What Are Design Patterns About?

At their core, design patterns separate the parts of your code that **change frequently** from those that **stay the same**. By isolating the changeable parts, you can modify them without affecting the stable parts. This makes your application easier to maintain and extend.

There are officially 23 design patterns, but they all aim to address this idea of separating the dynamic (changing) parts from the static (unchanging) ones. Today, we'll focus on the **Strategy Design Pattern** and see how it applies this principle.

2. What is the Strategy Design Pattern?

The Strategy Design Pattern is a way to define a family of interchangeable behaviors (or algorithms) and make them easy to swap at runtime. Instead of hardcoding behaviors into a class, you create separate classes for each behavior and let the main class use them flexibly.

Definition in Simple Terms

The Strategy Pattern lets you:

1. Define a set of behaviors (like walking, talking, or flying).
2. Put each behavior in its own class.
3. Allow the main class to switch between these behaviors dynamically.

This approach avoids the problems caused by using inheritance to handle different behaviors, which can lead to complex and rigid code.

3. Why Avoid Inheritance?

Inheritance is a core OOP concept; it allows a child class to inherit features from a parent class and add its own. However, relying heavily on inheritance can make your code hard to maintain, especially when new behaviors are introduced. Let's explore this with an example.

Example: Robot Simulation Application

Imagine we're building an application to simulate robots. Each robot has behaviors like walking, talking, and displaying its appearance (projection). We might start by creating `Robot` class with these behaviors as methods.

```
abstract class Robot {
    public void walk() {
        System.out.println("Robot is walking normally");
    }

    public void talk() {
        System.out.println("Robot is talking normally");
    }

    public abstract void projection();
}

class CompanionRobot extends Robot {
    @Override
    public void projection() {
        System.out.println("Companion Robot projection: Friendly and sleek");
    }
}

class WorkerRobot extends Robot {
    @Override
    public void projection() {
        System.out.println("Worker Robot projection: Sturdy and functional");
    }
}
```

The Problem with Inheritance

This setup works fine initially. `CompanionRobot` and `WorkRobot` inherit walk and talk from the `Robot` class and override the projection method to define their unique appearance.

But what happens when we add a new type of robot, say a `SparrowRobot`, that can also **fly**? We might add a fly method to its class:

```
class SparrowRobot extends Robot {
    @Override
    public void projection() {
        System.out.println("Sparrow Robot projection: Bird-like and agile");
    }

    public void fly() {
        System.out.println("Sparrow Robot is flying with wings");
    }
}
```

This works, but what if another robot, like a `CrowRobot`, also flies? We'd copy the same `fly` method into its class, violating the **DRY (Don't Repeat Yourself)** principle. Repeating code is inefficient and error-prone.

To avoid this, we might move the `fly` method to the `Robot` parent class:

```
abstract class Robot {
    public void walk() {
        System.out.println("Robot is walking normally");
    }

    public void talk() {
        System.out.println("Robot is talking normally");
    }

    public void fly() {
        System.out.println("Robot cannot fly");
    }

    public abstract void projection();
}
```

Now, all robots inherit the `fly` method, but most don't need it, and those that do (like `SparrowRobot`) might fly differently (e.g., with wings or jets). Overriding `fly` in multiple classes leads to the same DRY violation.

To fix this, we could create an inheritance hierarchy with a `FlyableRobot` class:

```java
abstract class FlyableRobot extends Robot {
    public void fly() {
        System.out.println("Flying with wings");
    }
}

class SparrowRobot extends FlyableRobot {
    @Override
    public void projection() {
        System.out.println("Sparrow Robot projection: Bird-like and agile");
    }
}

class CrowRobot extends FlyableRobot {
    @Override
    public void projection() {
        System.out.println("Crow Robot projection: Dark and swift");
    }
}
```

But what if a new robot, like `JetRobot`, flies with jets instead of wings? We'd need another subclass, `JetFlyableRobot`, and the hierarchy keeps growing. If we add more behaviors (e.g., robots that can't walk or talk), the inheritance tree becomes a tangled mess of permutations:

- Walkable vs. Non-Walkable
- Talkable vs. Non-Talkable
- Flyable vs. Non-Flyable
- FlyWithWings vs. FlyWithJet

This complexity violates the **Open-Closed Principle** (code should be open for extension but closed for modification) and makes maintenance a nightmare. The solution to inheritance is not more inheritance."

4. The Strategy Design Pattern Solution

The Strategy Pattern solves this by **favoring composition over inheritance**. Instead of using inheritance to define behaviors, we:

1. Extract behaviors (like walking, talking, flying) into separate interfaces.
2. Create concrete classes for each behavior variation.
3. Let the Robot class use these behaviors via composition (a "has-a" relationship).

Step-by-Step Implementation

Let's redesign the robot application using the Strategy Pattern.

1. Define Behavior Interfaces

We create interfaces for each behavior: `Walkable`, `Talkable`, and `Flyable`. Each interface has a single method.

```
interface Walkable {
    void walk();
}

interface Talkable {
    void talk();
}

interface Flyable {
    void fly();
}
```

2. Create Concrete Behavior Classes

For each interface, we create classes that implement different variations of the behavior.

```
class NormalWalk implements Walkable {
    @Override
    public void walk() {
        System.out.println("Walking normally");
    }
}

class NoWalk implements Walkable {
    @Override
    public void walk() {
        System.out.println("Cannot walk");
    }
}

class NormalTalk implements Talkable {
    @Override
    public void talk() {
        System.out.println("Talking normally");
    }
}

class NoTalk implements Talkable {
    @Override
    public void talk() {
        System.out.println("Cannot talk");
    }
}
```

```java
class NormalFly implements Flyable {
    @Override
    public void fly() {
        System.out.println("Flying with wings");
    }
}

class NoFly implements Flyable {
    @Override
    public void fly() {
        System.out.println("Cannot fly");
    }
}
```

3. Redesign the Robot Class

The `Robot` class now uses composition to hold references to these behaviors. It delegates behavior execution to the respective interfaces.

```java
abstract class Robot {
    Walkable walkBehavior;
    Talkable talkBehavior;
    Flyable flyBehavior;

    public Robot(Walkable walkBehavior, Talkable talkBehavior, Flyable flyBehavior) {
        this.walkBehavior = walkBehavior;
        this.talkBehavior = talkBehavior;
        this.flyBehavior = flyBehavior;
    }

    public void walk() {
        walkBehavior.walk();
    }

    public void talk() {
        talkBehavior.talk();
    }

    public void fly() {
        flyBehavior.fly();
    }

    public abstract void projection();
}
```

4. Create Robot Types

Each robot type (e.g., `CompanionRobot`, `WorkerRobot`) specifies its behaviors via the constructor.

```java
class CompanionRobot extends Robot {
    public CompanionRobot(Walkable walkBehavior, Talkable talkBehavior, Flyable flyBehavior) {
        super(walkBehavior, talkBehavior, flyBehavior);
    }

    @Override
    public void projection() {
        System.out.println("Companion Robot projection: Friendly and sleek");
    }
}

class WorkerRobot extends Robot {
    public WorkerRobot(Walkable walkBehavior, Talkable talkBehavior, Flyable flyBehavior) {
        super(walkBehavior, talkBehavior, flyBehavior);
    }

    @Override
    public void projection() {
        System.out.println("Worker Robot projection: Sturdy and functional");
    }
}
```

5. Using the Robots

In the main program, we create robots and assign their behaviors dynamically.

```java
public class Main {
    public static void main(String[] args) {
        Robot companion = new CompanionRobot(new NormalWalk(), new NormalTalk(), new NoFly());
        Robot worker = new WorkerRobot(new NoWalk(), new NoTalk(), new NormalFly());

        System.out.println("Companion Robot:");
        companion.walk();
        companion.talk();
        companion.fly();
        companion.projection();

        System.out.println("\nWorker Robot:");
        worker.walk();
        worker.talk();
        worker.fly();
        worker.projection();
    }
}
```

OUTPUT:

```
Companion Robot:
Walking normally
Talking normally
Cannot fly
Companion Robot projection: Friendly and sleek

Worker Robot:
Cannot walk
Cannot talk
Flying with wings
Worker Robot projection: Sturdy and functional
```

5. Eliminating Inheritance Completely

Notice that we're still using inheritance for the `Projection` method. To remove inheritance entirely, we can extract `Projection` into a `Projectable` interface:

```java
interface Projectable {
    void projection();
}

class CompanionProjection implements Projectable {
    @Override
    public void projection() {
        System.out.println("Companion Robot projection: Friendly and sleek");
    }
}

class WorkerProjection implements Projectable {
    @Override
    public void projection() {
        System.out.println("Worker Robot projection: Sturdy and functional");
    }
}
```

Then, update the `Robot` class to use composition for `Projection`:

```
class Robot {
    Walkable walkBehavior;
    Talkable talkBehavior;
    Flyable flyBehavior;
    Projectable projectionBehavior;

    public Robot(Walkable walkBehavior, Talkable talkBehavior, Flyable flyBehavior, Projectable projectionBehavior) {
        this.walkBehavior = walkBehavior;
        this.talkBehavior = talkBehavior;
        this.flyBehavior = flyBehavior;
        this.projectionBehavior = projectionBehavior;
    }

    public void walk() {
        walkBehavior.walk();
    }

    public void talk() {
        talkBehavior.talk();
    }

    public void fly() {
        flyBehavior.fly();
    }

    public void projection() {
        projectionBehavior.projection();
    }
}
```

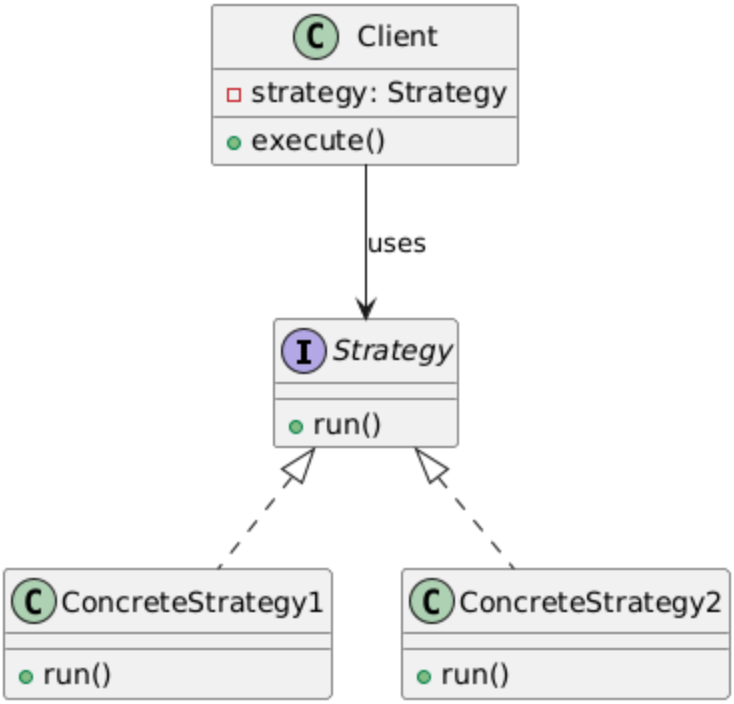
Now, we create robots without any inheritance:

```
public class Main {
    public static void main(String[] args) {
        Robot companion = new Robot(new NormalWalk(), new NormalTalk(), new NoFly(), new CompanionProjection());
        Robot worker = new Robot(new NoWalk(), new NoTalk(), new NormalFly(), new WorkerProjection());

        System.out.println("Companion Robot:");
        companion.walk();
        companion.talk();
        companion.fly();
        companion.projection();

        System.out.println("\nWorker Robot:");
        worker.walk();
        worker.talk();
        worker.fly();
        worker.projection();
    }
}
```

This approach eliminates inheritance entirely, making the design even more flexible.

6. UML Diagram for Strategy Pattern

Here's the UML diagram for the Strategy Pattern, showing the relationships between the client (`Robot`) and the strategy interfaces and their implementations.



Real-Life Examples of the Strategy Pattern

The Strategy Pattern is widely used in real applications. Here are two examples:

1. Payment System

Imagine an e-commerce application with a `PaymentSystem` class that processes payments. Payments can be made via jazzcash, credit/debit card, or net banking.

2. Sorting System

Consider a sorting application where you can sort data using different algorithms (e.g., QuickSort, MergeSort).

Here, in this case **SortingSystem** is the client, and QuickSort and MergeSort are strategies.

The source code for these is available in Python and C++ on my GitHub repository. You can access it via the following link: [Insert GitHub Link Here].

7. Key Benefits of the Strategy Pattern

1. **Flexibility**: Behaviors can be swapped at runtime without changing the client code.
2. **Reusability**: Behavior classes can be reused across different clients.

3. **Maintainability**: Adding new behaviors (e.g., FlyWithJet) requires only a new class, not changes to existing code.
4. **Adheres to SOLID Principles**: Supports the Open-Closed Principle by allowing extension without modification.
5. **Eliminates Code Duplication**: Avoids repeating behavior code across classes.

8. Conclusion

The Strategy Design Pattern is a powerful tool for making your code flexible and maintainable. By separating behaviors into their own classes and using composition, you can avoid the pitfalls of complex inheritance hierarchies. It's especially useful in applications where behaviors change frequently or need to be swapped dynamically.

Favor composition over inheritance. This principle, embodied by the Strategy Pattern, ensures your code is easier to extend and maintain.

Project: LLD Design - Bazaar Online Marketplace Application

This guide outlines the design of "Bazaar," an online marketplace app similar to Daraz, for an LLD interview. It covers requirement gathering, happy flow discussion, UML diagrams, and a modular C++ implementation, ensuring the system is scalable, maintainable, and aligned with user needs (e.g., shopping for shalwar kameez in Lahore or electronics in Karachi).

Step 1: Requirement Gathering

We start by defining **functional** and **non-functional requirements** to set the scope, as is typical in an LLD interview.

Functional Requirements

These define the core features of Bazaar:

- **User Actions**:
  - Users can search for products (e.g., shalwar kameez, smartphones) by category or location (e.g., Karachi, Lahore).
  - Users can browse seller listings and add products to a cart.
  - Users can place orders (delivery or pickup) and make payments using local methods (e.g., JazzCash, EasyPaisa, credit card).
  - Users receive notifications after order placement (e.g., via SMS or WhatsApp, common in Pakistan).
- **Entities**:
  - **User**: Represents the customer (e.g., with a CNIC-based ID).

    - **Seller**: Represents a vendor (e.g., a shop in Anarkali Bazaar, Lahore) with a product catalog.
    - **Product**: Represents items (e.g., "Samsung Galaxy A14," "Embroidered Shalwar Kameez") with details like name, price, and code.
    - **Cart**: Holds selected products from a single seller.
    - **Order**: Represents a placed order with details like user, seller, products, and payment method.
    - **Payment**: Supports local payment methods like JazzCash or credit card.
    - **Notification**: Sends confirmations for order placement.
- **Persistence**:
    - Save order details to a file (scalable to a database like MongoDB).

Non-Functional Requirements

These ensure system quality:

- **Scalability**: Handle multiple users and sellers across cities like Islamabad and Peshawar.
- **Loose Coupling**: Components (e.g., cart, payment) should be independent.
- **Extensibility**: Support new product categories (e.g., home appliances) or payment methods (e.g., bank transfer).
- **Maintainability**: Use clean code and SOLID principles (Single Responsibility, Open-Closed, etc.).
- **Performance**: Ensure fast product search and checkout, critical for busy markets like Saddar, Karachi.

Clarifications with Interviewer

To keep the scope manageable:

- Focus on user flow (not seller flow, e.g., adding products)?
    - Assume user flow only.
- Are payments handled by a third-party service?
    - Assume third-party integration (e.g., JazzCash API).
- Are notifications internal or external?
    - Assume external service integration (e.g., SMS/WhatsApp).
- Support for scheduled deliveries?
    - Yes, include immediate and scheduled deliveries.

This ensures alignment with the interviewer and keeps the design interview-friendly.

Step 2: Happy Flow Discussion

The **happy flow** outlines the user's journey through Bazaar:

1. **User Searches for Products**:
    - User enters a category (e.g., "Electronics") or location (e.g., "Lahore").

    - System returns a list of products from sellers in that category/location.
2. **User Browses and Selects**:
    - User selects a seller and views their product catalog.
    - User adds products to a cart (e.g., "Samsung Galaxy A14").
3. **User Places Order**:
    - User reviews the cart and chooses delivery or pickup (e.g., at Liberty Market, Lahore).
    - User selects a payment method (e.g., JazzCash) and completes the payment.
4. **Order Confirmation**:
    - System processes the order and sends a notification to the user.
    - Cart is cleared post-order.

This flow ensures clarity with the interviewer before designing the system.

Step 3: UML Diagram Design

We'll use a **bottom-up approach**, starting with smaller objects (e.g., Product) and building relationships to larger ones (e.g., Order), as it's effective for LLD interviews.

Classes and Relationships

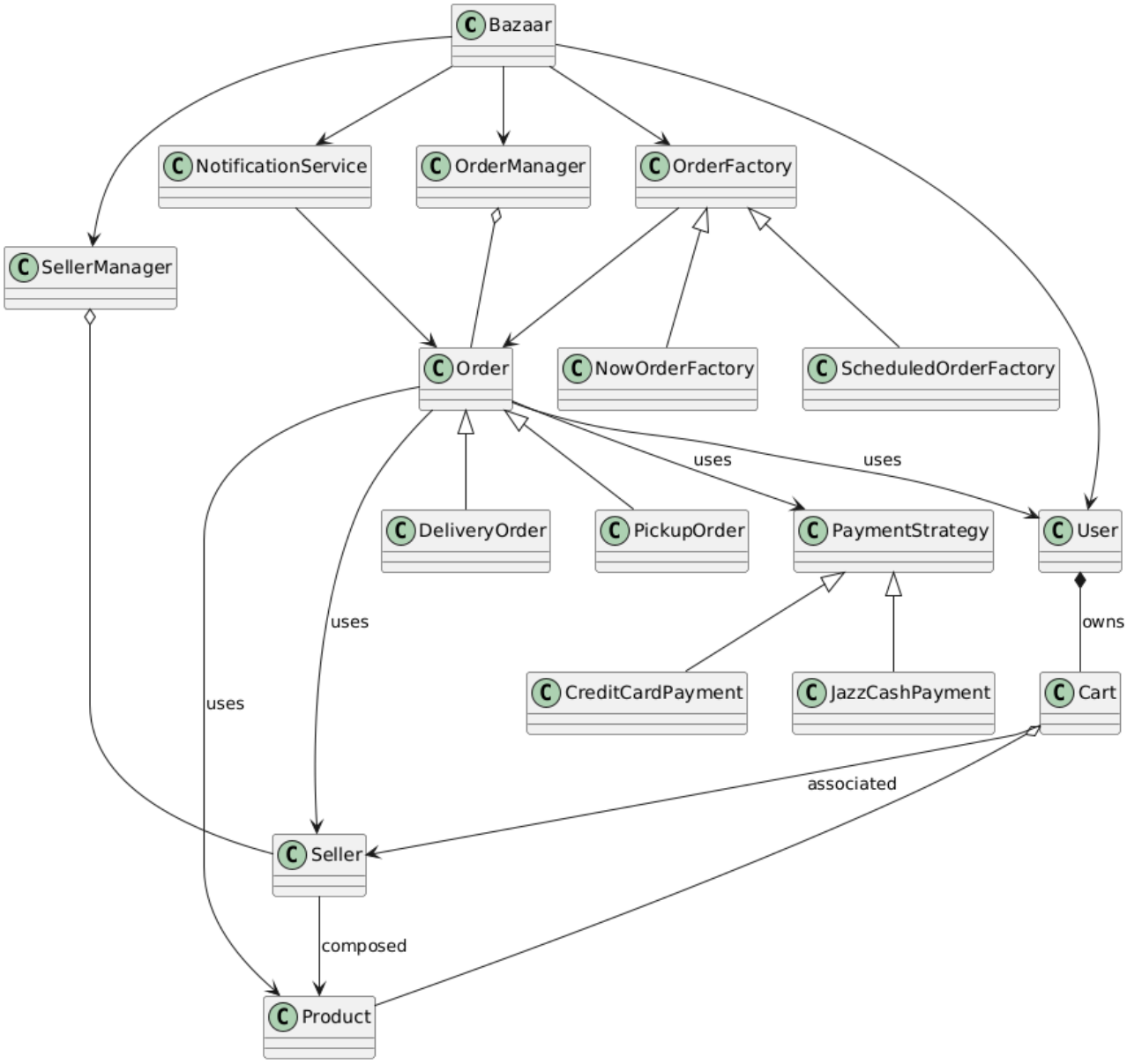
1. **Product (Model)**:
    - Attributes: `code`(string), `name` (string), `price` (double), `category` (string).
    - Methods: Getters and setters.
    - Purpose: Represents an item like "Embroidered Shalwar Kameez" or "Samsung Galaxy A14."
2. **Seller (Model)**:
    - Attributes: `SellerId`(string), `name` (string), `address` (string), `products` (`product`)
    - Methods: Getters and setters.
    - Relationship: **Composition** with `Product`(Products cannot exist without a Seller).
3. **SellerManager (Singleton)**:
    - Attributes: `sellers`(`seller`).
    - Methods:
        - `addSeller`(`seller`): Adds a new seller.
        - `searchByLocation`(String): Returns sellers based on location.
        - `searchByCategory`(String): Returns sellers with products in a category.
    - Purpose: Manages seller CRUD operations and search functionality.
    - Relationship: **Aggregation** with Seller.
4. **User (Model)**:
    - Attributes: `UserId`(string), `name`(string), `address`(string), `cart`(Cart).
    - Methods: Getters and setters.
    - Relationship: **Composition** with `Cart`.

5. **Cart (Model)**:
    - Attributes: `seller`(Seller), `products`(`Product`), `totalPrice` (double).
    - Methods:
        - `addProduct`(`Product`): Adds a product to the cart.
        - `Clear()`: Empties the cart.
        - `isEmpty()`: Checks if the cart is empty.
        - `getTotalCost()`: Calculates total price.
    - Relationships:
        - **Association** with `seller`(Cart references one Seller).
        - **Aggregation** with `Product`.
6. **Order (Abstract Model)**:
    - Attributes: `orderId`(string), `user`(User), `seller`(Seller), `products`(`Product`),`paymentStrategy(PaymentStrategy)`, `total` (double), `scheduleTime` (string).
    - Methods:
        - `processPayment()`: Processes payment using the strategy.
        - `getType()`(abstract): Returns order type (overridden by subclasses).
    - Relationships:
        - **Association** with `User`, `Seller`, `Product`, `PaymentStrategy`.
    - Subclasses:
        - **DeliveryOrder**: Adds `userAddress`(string).
        - **PickupOrder**: Adds `sellAddress`(string).
    - Purpose: Represents an order with delivery or pickup types.
7. **OrderFactory (Interface)**:
    - Method: `createOrder(User, Cart ,Seller, list<Product>, PaymentStrategy, string orderType, double totalCost`.
    - Subclasses:
        - **NowOrderFactory**: Creates immediate orders.
        - **ScheduledOrderFactory**: Creates scheduled orders.
    - Purpose: Implements the **Factory Method Pattern** for order creation.
    - Relationship: **Creates** `Order`.
8. **OrderManager (Singleton)**:
    - Attributes: `orders`(List<order>).</order>
    - Methods:
        - `addOrder(Order)`: Adds an order.
        - `listOrders()`: Returns all orders.
    - Purpose: Manages order storage and retrieval.
    - Relationship: **Aggregation** with `Order`.
9. **PaymentStrategy (Interface)**:
    - Method: `pay(double amount)`.
    - Subclasses:
        - **JazzCashPayment**: Uses mobile number for payment.
        - **CreditCardPayment**: Uses card number.
    - Purpose: Implements the **Strategy Pattern** for flexible payment methods.
    - Relationship: Used by `Order`.
10. **NotificationService**:
    - Method: `notify(order)`:Sends notification for an order.

- Purpose: Integrates with a third-party service (e.g., SMS/WhatsApp).
- Relationship: **Association** with `Order`.

11. **Bazaar (Orchestrator)**:
    - Methods:
        - `searchProductsByCategory(String category)`: Delegates to `SellerManager`.
        - `searchProductsByLocation(String location)`: Delegates to `SellerManager`.
        - `selectSeller(User, Seller)`: Assigns a seller to the user's cart.
        - `addToCart(User, String productCode)`: Adds a product to the cart.
        - `checkoutNow(User, PaymentStrategy, String orderType)`: Creates an immediate order.
        - `checkoutScheduled(User, PaymentStrategy, String orderType, String scheduledTime)`: Creates a scheduled order.
        - `payForOrder(Order)`: Processes payment and sends notification.
        - `printUserCart(User)`: Displays cart contents.
    - Purpose: Orchestrates interactions with other components, acting as a facade for the client (e.g., frontend).
    - Relationships: Interacts with `SellerManager`, `OrderManager`, `Cart`, `Order`, `NotificationService`.

UML Diagram



- **Composition**: `Seller → Product, User → Cart.`
- **Aggregation**: `SellerManager → Seller, OrderManager → Order, Cart → Product.`
- **Association**: `Cart → Seller, Order → User, Seller, Product, PaymentStrategy.`
- **Inheritance**: `Order → DeliveryOrder, PickupOrder; PaymentStrategy → JazzCashPayment, CreditCardPayment; OrderFactory → NowOrderFactory, ScheduledOrderFactory.`

Step 4: Code Implementation

Below is a modular C++ implementation of Bazaar, organized into folders for clarity (models, managers, factories, strategies, services, utils). The code adheres to SOLID principles and uses design patterns (Singleton, Factory Method, Strategy).

The source code for these is available in C++ on my GitHub repository. You can access it via the following link: [https://github.com/MuhammadTahaNasir/Systems_Design-Codes-Notes/tree/main/Week-2].

Step 5: Design Patterns Used

1. **Singleton (SellerManager, OrderManager)**:
    - Ensures a single instance for managing sellers and orders, maintaining consistent state.
2. **Factory Method (OrderFactory)**:
    - Encapsulates order creation logic for immediate and scheduled orders.
3. **Strategy (PaymentStrategy)**:
    - Supports flexible payment methods (JazzCash, Credit Card) with easy extensibility (e.g., for EasyPaisa).
4. **Orchestrator (Bazaar)**:
    - Acts as a facade, centralizing client interactions, though it slightly violates Single Responsibility for simplicity.

Step 6: Potential Extensions and Trade-offs

1. **PaymentStrategyFactory**:
    - Introduce a factory to dynamically select payment strategies, reducing client responsibility in main.cpp.
2. **NotificationService Hierarchy**:
    - Make NotificationService an interface with implementations for SMS, WhatsApp, or Urdu notifications.
3. **Decentralized Approach**:
    - Replace Bazaar orchestrator with API/service layers (e.g., FastAPI endpoints) for better modularity.
    - Example: /searchProducts/{category}, /checkout/{userId} APIs.
4. **Database Integration**:
    - Store sellers, orders, and users in MongoDB/PostgreSQL, managed by SellerManager and OrderManager.
5. **Additional Features**:
    - Support for product ratings, discounts, or AI-based price recommendations (e.g., using TensorFlow).
    - Add support for Urdu product descriptions or location-based filtering (e.g., "Shops near Gulberg").

Trade-offs

- **Bazaar Orchestrator**:
  - **Pros**: Simplifies client interaction with a single-entry point.
  - **Cons**: Slightly violates Single Responsibility and Least Knowledge due to multiple dependencies.
  - **Alternative**: Use API/service layers, increasing complexity but improving modularity.
- **OrderFactory Parameters**:
  - **Pros**: Passing all parameters ensures loose coupling.
  - **Cons**: Increases method complexity.
  - **Alternative**: Pass only User and derive details, but this couples OrderFactory to User and Cart internals.

Step 7: Key Takeaways for LLD Interviews

1. **Clarify Requirements**: Ask questions to narrow scope and align with the interviewer.
2. **Discuss Happy Flow**: Validate the user journey to ensure clarity.
3. **Use UML Diagrams**: Visualize relationships and confirm with the interviewer before coding.
4. **Apply Design Patterns**: Use Singleton, Factory, and Strategy to demonstrate OOP expertise.
5. **Write Clean Code**: Modularize code, adhere to SOLID principles, and handle edge cases.
6. **Engage the Interviewer**: Discuss trade-offs and alternatives collaboratively.
