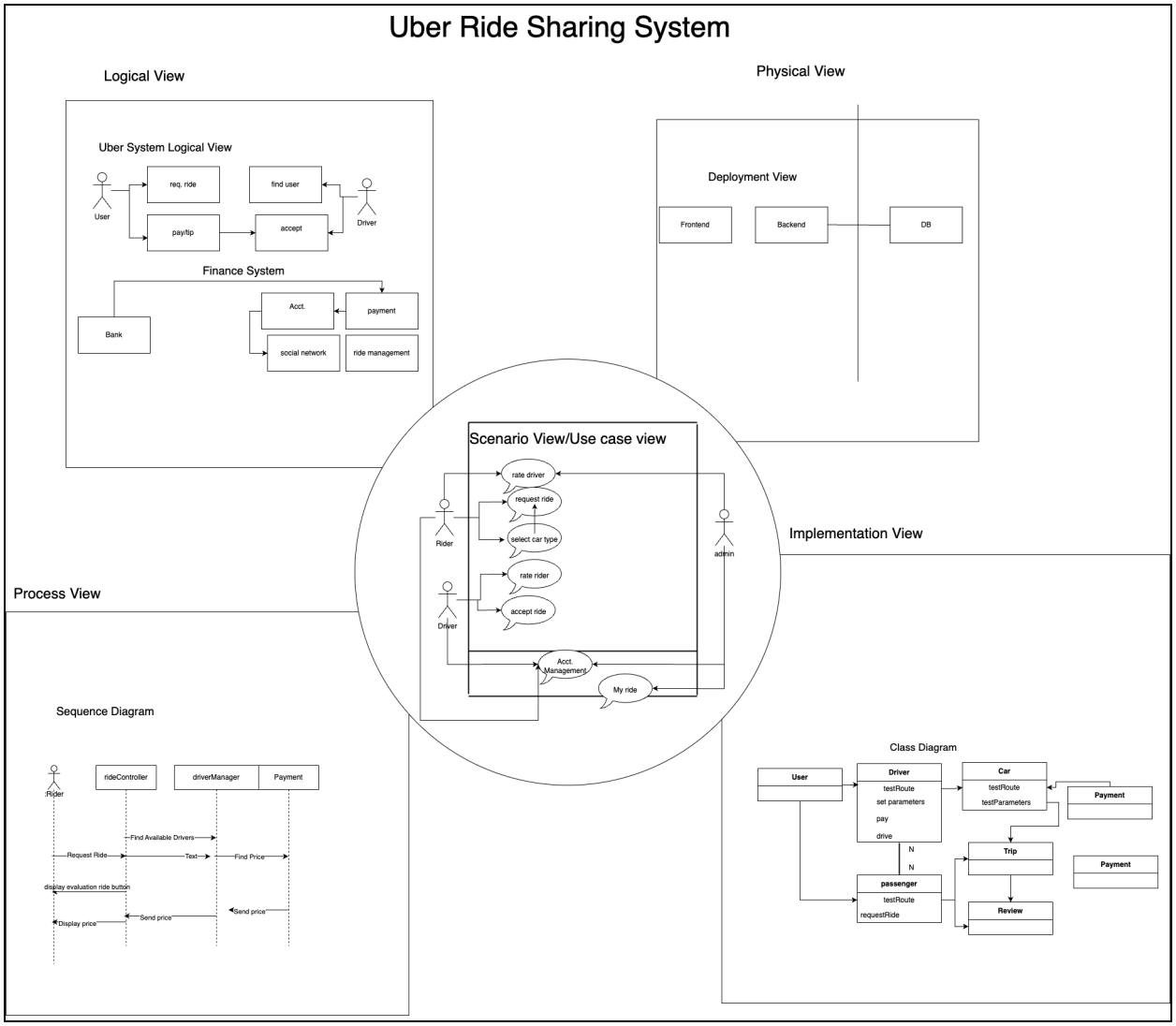
Due 4/2

Goal: In this assignment, you will practice applying design patterns in software development. Specially, you will practice using design patterns on a sample final exam.

Each team will work together.

Here is a simplified scenario of a software system supporting uber-like car sharing services. Your company developed a taxi reservation system before, and you are expected to design the new system as a software architect.

MAX: [18] 1. As an architect, you should provide a 4+1 View design for the system using the corresponding UML diagrams, e.g., scenario view (use case diagram); logical view (component diagram); development view (class diagram); progress view (sequence diagram); and physical view (deployment diagram).



GRANT [12] 2. Your system will offer three basic service levels. carPOOL is the least expensive level of service, in which the customer may share the ride with another passenger going in the same general direction. carX is a level of service in which the rider will get a private ride. carBlack is a level of service in which the rider is provided a black luxury car. Each such service will be handled by dedicated subsystems, but you surely will not let customers know all such details. Instead, you would like to provide an interface for customers to select from, and based on their selections, you will direct them to corresponding modules to calculate their transportation fees. Two design patterns may be used here together. Briefly explain your recommendation, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For each design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

A) CarPool - share ride, cheapest

B) CarX - private ride
C) CarBlack - black luxury car

Interfaces → modules (fees)

Answer : 2 Design patterns together. Explain recommendation.

I want to use **Factory Method + Strategy** combination which I think works well because:

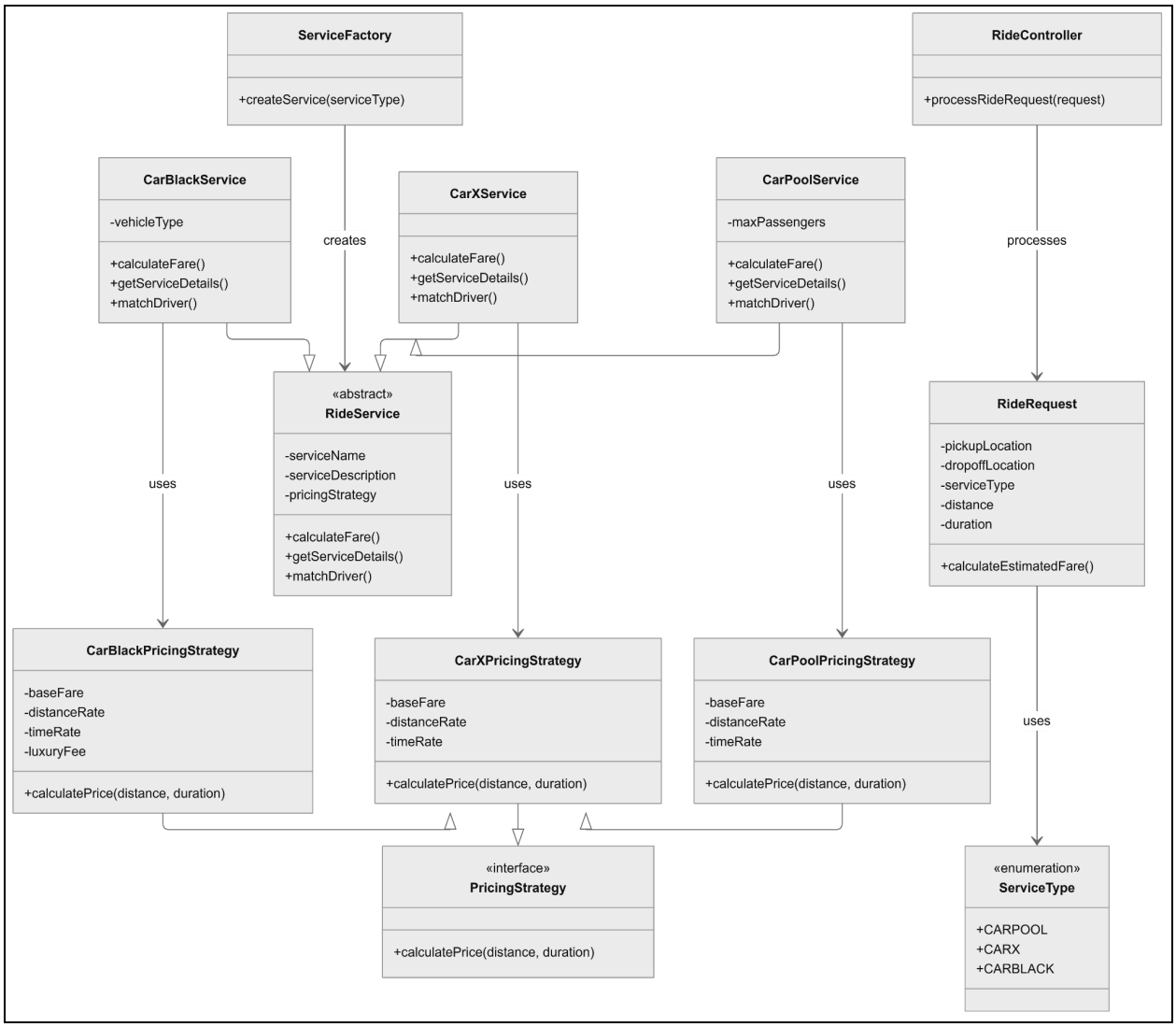
1. Factory Method handles the creation of appropriate service objects based on customer selection, abstracting away the details of which concrete service is being used.
2. Strategy handles the different pricing calculations for each service level and each service level can have its own pricing algorithm.
3. Together they provide good separation of concerns and hide implementation details.

This combination directly addresses our requirements:

- Factory Method hides the implementation details from customers
- Strategy allows for different pricing calculations
- Both patterns together make the system extensible for future service levels

The Factory Method will create the appropriate service object (carPOOL, carX, or carBlack) based on customer selection, and each service will utilize its own pricing strategy. The customer only needs to interact with a simple interface without knowing the underlying complexities. This approach is clean, maintainable, and directly addresses the requirements of the system. It also allows for adding new service levels or modifying pricing strategies in the future without disrupting the existing code.

Answer : Draw class diagram.



Class Descriptions:

Factory Method Pattern Classes:

1. **ServiceType (Enum)**:
    - Defines the available service types (CARPOOL, CARX, CARBLACK)
    - Used by clients to specify which service they want without knowing implementation details
2. **ServiceFactory**:
    - Creates the appropriate RideService object based on the requested ServiceType
    - Encapsulates the instantiation logic, hiding complexity from clients
    - Single point of modification when adding new service types

3. **RideService (Abstract)**:
    - Base class for all service implementations
    - Defines common interface and implements shared functionality
    - Contains a reference to a pricing strategy
4. **CarPoolService**:
    - Implements shared ride service with multiple passengers
    - Optimized for cost-effectiveness at the expense of privacy
    - Tracks maximum number of passengers that can share a ride
5. **CarXService**:
    - Implements standard private ride service
    - Balanced offering between economy and luxury
6. **CarBlackService**:
    - Implements premium luxury ride service
    - Uses high-end vehicles with additional amenities
    - Includes special vehicle type information

Strategy Pattern Classes:

1. **PricingStrategy (Interface)**:
    - Defines the contract for all pricing algorithms
    - Allows different pricing calculations to be swapped at runtime
2. **CarPoolPricingStrategy**:
    - Implements economy pricing with low base fare and rates
    - May include special discounts for shared rides
3. **CarXPricingStrategy**:
    - Implements standard pricing with medium base fare and rates
4. **CarBlackPricingStrategy**:
    - Implements premium pricing with high base fare and rates
    - Includes additional luxury fee
5. **RideRequest**:
    - Encapsulates all data needed for a ride request
    - Acts as a data transfer object between client and system
6. **Client**:
    - Represents the user interface that interacts with the system
    - Uses the factory to create services without knowing concrete implementations

Answer: Integrated Design Pattern Sample Code:

```
enum ServiceType {
```

```
    CARPOOL, CARX, CARBLACK
}

// Strategy interface
interface PricingStrategy {
    double calculatePrice(double distance, double duration);
}

// Concrete strategies
class CarPoolPricingStrategy implements PricingStrategy {
    public double calculatePrice(double distance, double duration) {
        return 2.0 + (distance * 0.5) + (duration * 0.1);
    }
}

class CarXPricingStrategy implements PricingStrategy {
    public double calculatePrice(double distance, double duration) {
        return 5.0 + (distance * 1.0) + (duration * 0.2);
    }
}

class CarBlackPricingStrategy implements PricingStrategy {
    public double calculatePrice(double distance, double duration) {
```

```
        return 10.0 + (distance * 2.0) + (duration * 0.5);
    }
}

// Abstract RideService
abstract class RideService {
    protected String serviceName;
    protected PricingStrategy pricingStrategy;

    public double calculateFare(double distance, double duration) {
        return pricingStrategy.calculatePrice(distance, duration);
    }

    public String getServiceName() {
        return serviceName;
    }
}

// Concrete services
class CarPoolService extends RideService {
    public CarPoolService() {
        this.pricingStrategy = new CarPoolPricingStrategy();
        this.serviceName = "carPOOL";
```

```
    }
}

class CarXService extends RideService {
    public CarXService() {
        this.pricingStrategy = new CarXPricingStrategy();
        this.serviceName = "carX";
    }
}

class CarBlackService extends RideService {
    public CarBlackService() {
        this.pricingStrategy = new CarBlackPricingStrategy();
        this.serviceName = "carBlack";
    }
}

// Factory
class ServiceFactory {
    public static RideService createService(ServiceType type) {
        return switch (type) {
            case CARPOOL -> new CarPoolService();
            case CARX    -> new CarXService();
```

```
        case CARBLACK-> new CarBlackService();
      };
  }
}

// Client code
public class Main {
  public static void main(String[] args) {
    RideService pool = ServiceFactory.createService(ServiceType.CARPOOL);
    System.out.println(pool.getServiceName() + " fare: $" + pool.calculateFare(10, 20));

    RideService black = ServiceFactory.createService(ServiceType.CARBLACK);
    System.out.println(black.getServiceName() + " fare: $" + black.calculateFare(10, 20));
  }
}

// Factory
class ServiceFactory:
  static method createService(type: ServiceType): RideService
    switch(type):
      case CARPOOL: return new CarPoolService()
      case CARX: return new CarXService()
      case CARBLACK: return new CarBlackService()
```

```
        default: throw Error("Unknown service type")


// Client code

function main():

    // Client uses factory to get service with appropriate strategy

    poolService = ServiceFactory.createService(CARPOOL)

    print(poolService.getServiceName() + " fare: $" + poolService.calculateFare(10, 20))


    blackService = ServiceFactory.createService(CARBLACK)

    print(blackService.getServiceName() + " fare: $" + blackService.calculateFare(10, 20))
```

GRANT [7] 3. Your system may offer more service levels at some cities or countries. For example, you may want to offer a service carGo in India, which provides for a ride in a hatchback. For another example, carEATS will allow users to have meals delivered from participating restaurants by your registered drivers. You want your system to be designed to provide such car sharing services in many cities and countries, meaning that although all cities will provide the basic levels of services, each city may configure her own customized services. Which design pattern will you recommend to use? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Each city needs custom car sharing services:

1. carGo in india for hatchback
2. carEATS for food delivery with registered restaurants

Answer: Which design pattern? Explain Recommendation

I want to use **Decorator Design Pattern** which I think works well because:

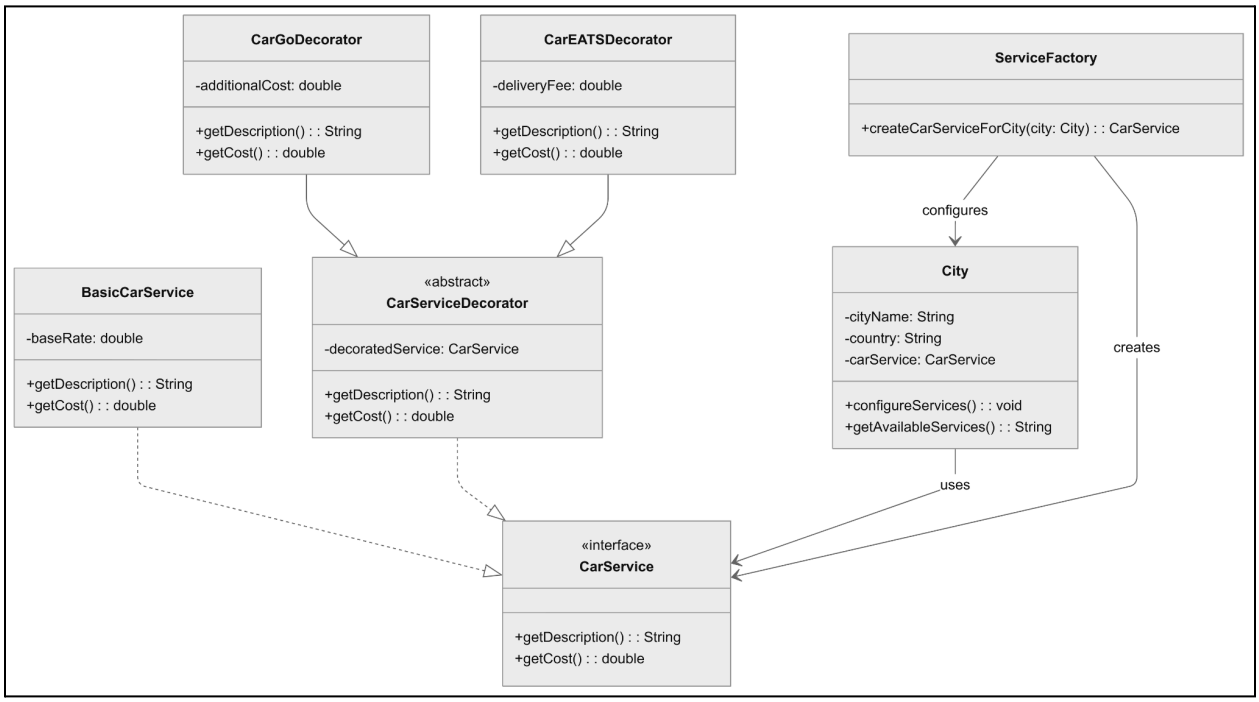
1. It allows attaching additional responsibilities to objects dynamically without modifying their structure

2. It follows the open/closed principle so the system is extensible for new services
3. It promotes composition over inheritance, which is ideal for combining different service features
4. It keeps individual components focused on single responsibilities, improving maintainability

This design pattern directly addresses our requirements:

- Basic car services can be implemented in a core component, while city-specific services are added as decorators
- New service types (like carGo or carEATS) can be created as separate decorators without affecting existing code
- Each city can configure its own unique combination of services by applying different decorators
- The system remains flexible for future expansion to new locations with their own service requirements

Answer : Draw class diagram



Class Descriptions:

CarService (Interface)

- Defines the common interface for all car service components

- Acts as the Component in the Decorator pattern
- Provides the contract that all concrete services and decorators must implement

BasicCarService

- Implements the basic ride-sharing functionality available in all cities
- Serves as the Concrete Component in the Decorator pattern
- Provides the core service that can be extended with additional features

CarServiceDecorator (Abstract)

- Maintains a reference to a CarService object it decorates
- Acts as the base Decorator in the pattern
- Implements CarService interface and delegates operations to the wrapped component

CarGoDecorator

- Adds hatchback ride service specific to India
- Represents a Concrete Decorator in the pattern
- Extends the basic service with specialized small vehicle options

CarEATSDecorator

- Adds food delivery capability with restaurant partners
- Represents another Concrete Decorator in the pattern
- Extends the basic service with an entirely different service dimension

City

- Represents a location with its specific service configuration
- Acts as the client context that uses decorated services
- Manages which service combinations are available in each location

ServiceFactory

- Creates appropriate service configurations for specific cities
- Encapsulates the knowledge of which decorators to apply based on location
- Simplifies client code by handling the decoration process

Decorator Pattern Application

- Enables dynamic addition of responsibilities to objects

- Allows each city to have a unique combination of services
- Provides a flexible alternative to subclassing for extending functionality
- Supports composition of decorators to create complex service offerings

Answer: Decorator Design Pattern Pseudocode

```
// Component Interface
interface CarService {
  String getDescription();
  double getCost();
}

// Concrete Component
class BasicCarService implements CarService {
  private final double baseRate = 5.0;

  public String getDescription() {
    return "Basic Car Service";
  }

  public double getCost() {
    return baseRate;
  }
}

// Abstract Decorator
abstract class CarServiceDecorator implements CarService {
  protected CarService decoratedCarService;
```

```
    public CarServiceDecorator(CarService carService) {
        this.decoratedCarService = carService;
    }

    public String getDescription() {
        return decoratedCarService.getDescription();
    }

    public double getCost() {
        return decoratedCarService.getCost();
    }
}

// Concrete Decorator 1
class CarGoDecorator extends CarServiceDecorator {
    private final double additionalCost = 2.0;

    public CarGoDecorator(CarService carService) {
        super(carService);
    }

    public String getDescription() {
        return super.getDescription() + ", with CarGo (Hatchback)";
    }
```

```
    public double getCost() {
        return super.getCost() + additionalCost;
    }
}

// Concrete Decorator 2
class CarEATSDecorator extends CarServiceDecorator {
    private final double deliveryFee = 3.5;

    public CarEATSDecorator(CarService carService) {
        super(carService);
    }

    public String getDescription() {
        return super.getDescription() + ", with CarEATS (Food Delivery)";
    }

    public double getCost() {
        return super.getCost() + deliveryFee;
    }
}
```

// Context Class

```
class City {
```

```
    private String cityName, country;
    private CarService carService;

    public City(String cityName, String country) {
        this.cityName = cityName;
        this.country = country;
    }

    public void configureServices() {
        carService = new BasicCarService();

        if (country.equals("India")) {
            carService = new CarGoDecorator(carService);
        }

        if (cityName.equals("Mumbai") || cityName.equals("New York")) {
            carService = new CarEATSDecorator(carService);
        }
    }

    public String getAvailableServices() {
        return cityName + ", " + country + ": " + carService.getDescription() +
                " (Cost: $" + carService.getCost() + ")";
    }
}
```

```
// Client code
public class Main {
    public static void main(String[] args) {
        City mumbai = new City("Mumbai", "India");
        mumbai.configureServices();
        System.out.println(mumbai.getAvailableServices());

        City tokyo = new City("Tokyo", "Japan");
        tokyo.configureServices();
        System.out.println(tokyo.getAvailableServices());

        City newYork = new City("New York", "USA");
        newYork.configureServices();
        System.out.println(newYork.getAvailableServices());
    }
}
```

---

[7] 4. **RAHIM** Your system will take payment through credit cards. But you do not want to implement your own credit card payment system; instead, you would leverage PayPal service for now. You decide to implement a handler locally in your code which prepares PayPal-compatible data format and forward the call to PayPal. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

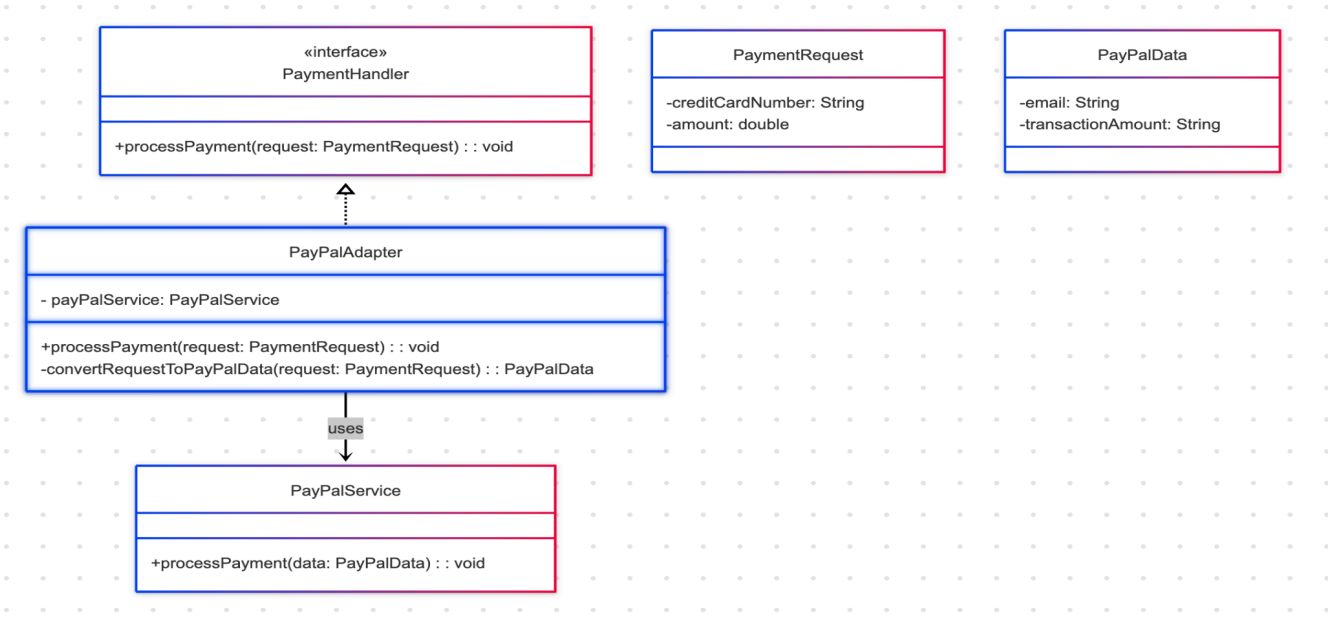
Leverage paypal service

Handler locally in code to prepare paypal compatible data format & forward call to Paypal

Appropriate design pattern? Justify

The Adapter Pattern is appropriate because it allows you to integrate an external payment service (PayPal) into your system without modifying the existing code that generates payment requests. By implementing a local handler (adapter) that implements your own payment interface, you convert the internal data format (credit card payment request) into the PayPal-specific format and delegate the call to the PayPal service. This maintains separation of concerns and adheres to the open/closed principle.

Class diagram for design pattern



Sample code for design pattern

```
// The interface used by your system for processing payments
public interface PaymentHandler {
    void processPayment(PaymentRequest request);
}
```

```
// Class representing a payment request in your system's native format (e.g., credit card details)
public class PaymentRequest {
    private String creditCardNumber;
    private double amount;

    public PaymentRequest(String creditCardNumber, double amount) {
        this.creditCardNumber = creditCardNumber;
        this.amount = amount;
    }

    public String getCreditCardNumber() {
        return creditCardNumber;
    }

    public double getAmount() {
        return amount;
    }
}

// Class representing the data format required by PayPal
public class PayPalData {
    private String email;
    private String transactionAmount;

    public PayPalData(String email, String transactionAmount) {
```

```
        this.email = email;
        this.transactionAmount = transactionAmount;
    }

    public String getEmail() {
        return email;
    }

    public String getTransactionAmount() {
        return transactionAmount;
    }
}

// Simulated PayPal service with its own interface
public class PayPalService {
    public void processPayment(PayPalData data) {
        System.out.println("Processing payment through PayPal:");
        System.out.println("Email: " + data.getEmail());
        System.out.println("Amount: " + data.getTransactionAmount());
    }
}

// Adapter that converts PaymentRequest to PayPalData and calls the PayPal service
public class PayPalAdapter implements PaymentHandler {
    private PayPalService payPalService;
```

```
    public PayPalAdapter(PayPalService service) {
        this.payPalService = service;
    }

    @Override
    public void processPayment(PaymentRequest request) {
        // Convert the PaymentRequest into PayPalData format
        PayPalData data = convertRequestToPayPalData(request);
        // Delegate to the PayPal service
        payPalService.processPayment(data);
    }

    // Conversion logic (could involve more complex mapping in a real scenario)
    private PayPalData convertRequestToPayPalData(PaymentRequest request) {
        // For demonstration: generate a dummy email from the credit card number
        String email = request.getCreditCardNumber() + "@payments.com";
        // Format the transaction amount as a string with two decimals
        String transactionAmount = String.format("%.2f", request.getAmount());
        return new PayPalData(email, transactionAmount);
    }
}

// Client code to test the implementation
public class PaymentClient {
```

```
    public static void main(String[] args) {
        // Create an instance of the PayPal service
        PayPalService payPalService = new PayPalService();
        // Create an adapter to handle payment requests for PayPal
        PaymentHandler paymentHandler = new PayPalAdapter(payPalService);
        // Create a PaymentRequest with credit card data
        PaymentRequest request = new PaymentRequest("1234-5678-9012-3456", 99.99);
        // Process the payment using the adapter
        paymentHandler.processPayment(request);
    }
}
```

[7] 5. **RAHIM** You decide to adopt a dynamic pricing model. The same route costs different amounts at different times as a result of factors such as the supply and demand for drivers at the time the ride is requested. When rides are in high demand in a certain area and there are not enough drivers in such area, fares increase to get more drivers to that area and to reduce demand for rides in that area. Therefore, you will calculate the transportation fee for a rider who will provide detailed information for you to calculate. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Dynamic Pricing Model

For our dynamic pricing model, the same route's price varies based on real-time conditions:

High rides + low drivers = Price Increase (Surge Pricing)
Low rides + high drivers = Price Decrease (Normal Pricing)

When a rider submits their ride details (such as distance, base fare, demand factor, and supply factor), the system must calculate the transportation fee dynamically based on these parameters.

Design Pattern for Calculating Rider Price

We recommend using the Strategy Pattern for this purpose.

Justification of Chosen Design Pattern

The Strategy Pattern is ideal because it:

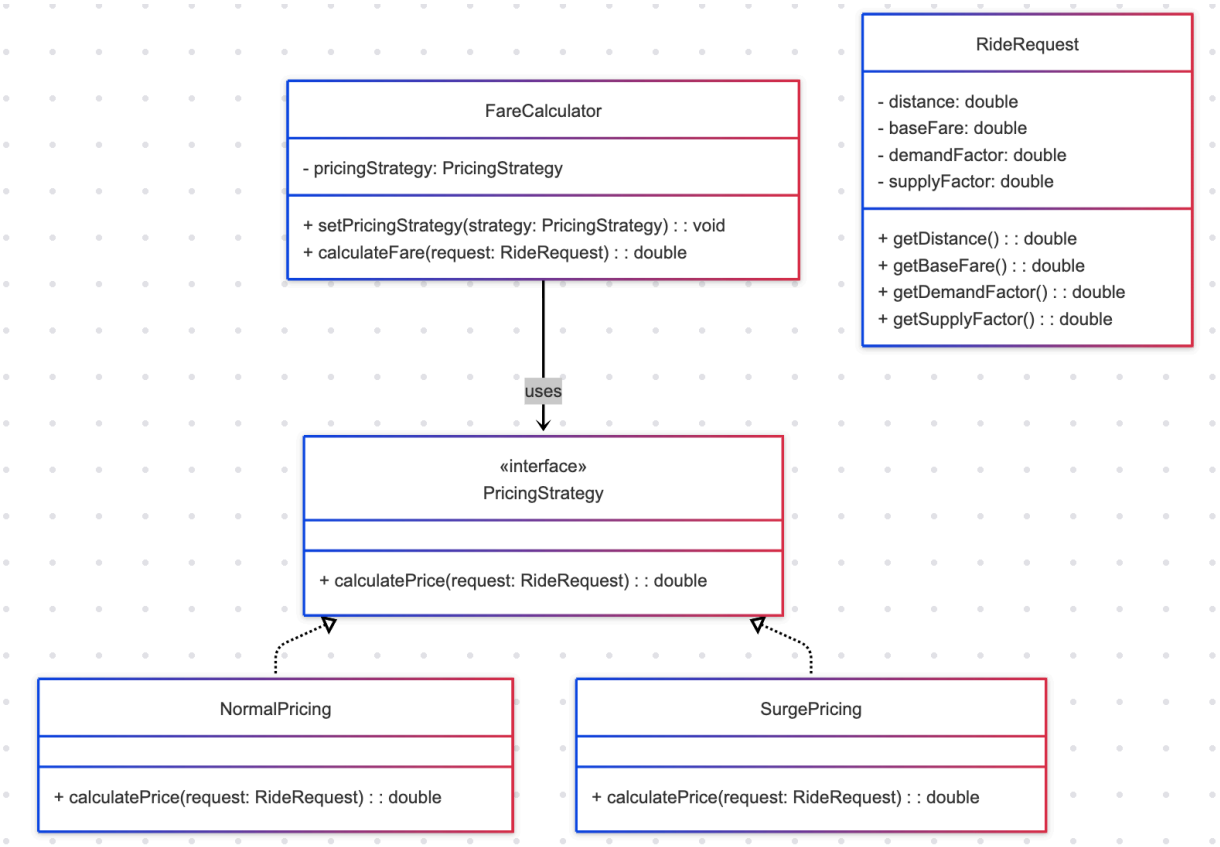
Encapsulates Multiple Algorithms: Each pricing method (normal or surge) is implemented in its own class, all following a common interface.

Allows Dynamic Selection: The appropriate pricing strategy can be chosen at runtime based on current supply/demand conditions.

Enhances Extensibility: Adding or modifying pricing algorithms is straightforward, as new strategies can be introduced without altering existing client code.

Promotes Separation of Concerns: Each pricing strategy is responsible for its own logic, making the system easier to maintain and test.

Class diagram



Sample code

```
// Context: FareCalculator that delegates to a PricingStrategy
public class FareCalculator {
  private PricingStrategy pricingStrategy;

  // Constructor to set an initial strategy
  public FareCalculator(PricingStrategy strategy) {
    this.pricingStrategy = strategy;
  }

  // Setter to change strategy dynamically at runtime
  public void setPricingStrategy(PricingStrategy strategy) {
```

```
        this.pricingStrategy = strategy;
    }

    // Calculate fare by delegating to the current strategy
    public double calculateFare(RideRequest request) {
        return pricingStrategy.calculatePrice(request);
    }
}

// Strategy interface defining the pricing method
public interface PricingStrategy {
    double calculatePrice(RideRequest request);
}

// Concrete strategy for normal pricing
public class NormalPricing implements PricingStrategy {
    @Override
    public double calculatePrice(RideRequest request) {
        // Basic fare calculation: base fare + distance * rate (e.g., 1.0)
        return request.getBaseFare() + (request.getDistance() * 1.0);
    }
}

// Concrete strategy for surge pricing (increases fare when demand > supply)
public class SurgePricing implements PricingStrategy {
```

```
    @Override
    public double calculatePrice(RideRequest request) {
        double multiplier = (request.getDemandFactor() > request.getSupplyFactor()) ? 1.5 :
1.0;
        return (request.getBaseFare() + request.getDistance()) * multiplier;
    }
}

// Data class representing the details of a ride request
public class RideRequest {
    private double distance;
    private double baseFare;
    private double demandFactor;
    private double supplyFactor;

    public RideRequest(double distance, double baseFare, double demandFactor, double
supplyFactor) {
        this.distance = distance;
        this.baseFare = baseFare;
        this.demandFactor = demandFactor;
        this.supplyFactor = supplyFactor;
    }

    public double getDistance() {
        return distance;
    }
```

```
    public double getBaseFare() {
        return baseFare;
    }

    public double getDemandFactor() {
        return demandFactor;
    }

    public double getSupplyFactor() {
        return supplyFactor;
    }
}

// Client code to test the dynamic pricing model
public class PricingClient {
    public static void main(String[] args) {
        // Create a ride request under normal conditions (low demand)
        RideRequest normalRequest = new RideRequest(10.0, 2.0, 1.0, 1.5);
        // Create a ride request under surge conditions (high demand)
        RideRequest surgeRequest = new RideRequest(10.0, 2.0, 2.0, 1.0);

        // Use NormalPricing strategy initially
        FareCalculator calculator = new FareCalculator(new NormalPricing());
        double normalFare = calculator.calculateFare(normalRequest);
```

```
        System.out.println("Normal pricing fare: " + normalFare);


        // Switch to SurgePricing strategy for high demand conditions
        calculator.setPricingStrategy(new SurgePricing());
        double surgeFare = calculator.calculateFare(surgeRequest);
        System.out.println("Surge pricing fare: " + surgeFare);
    }
}
```

[7] 6. **RAHIM** Your system may periodically send coupons to all registered riders automatically. Will you recommend to use any design pattern to realize this push-mode notification? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Design Pattern for Coupon Push-Mode Notification

When the system needs to periodically push coupons to all registered riders, the Observer pattern is an excellent choice. It allows the coupon sender (the subject) to notify all subscribed riders (observers) automatically whenever a new coupon is available.

Justification

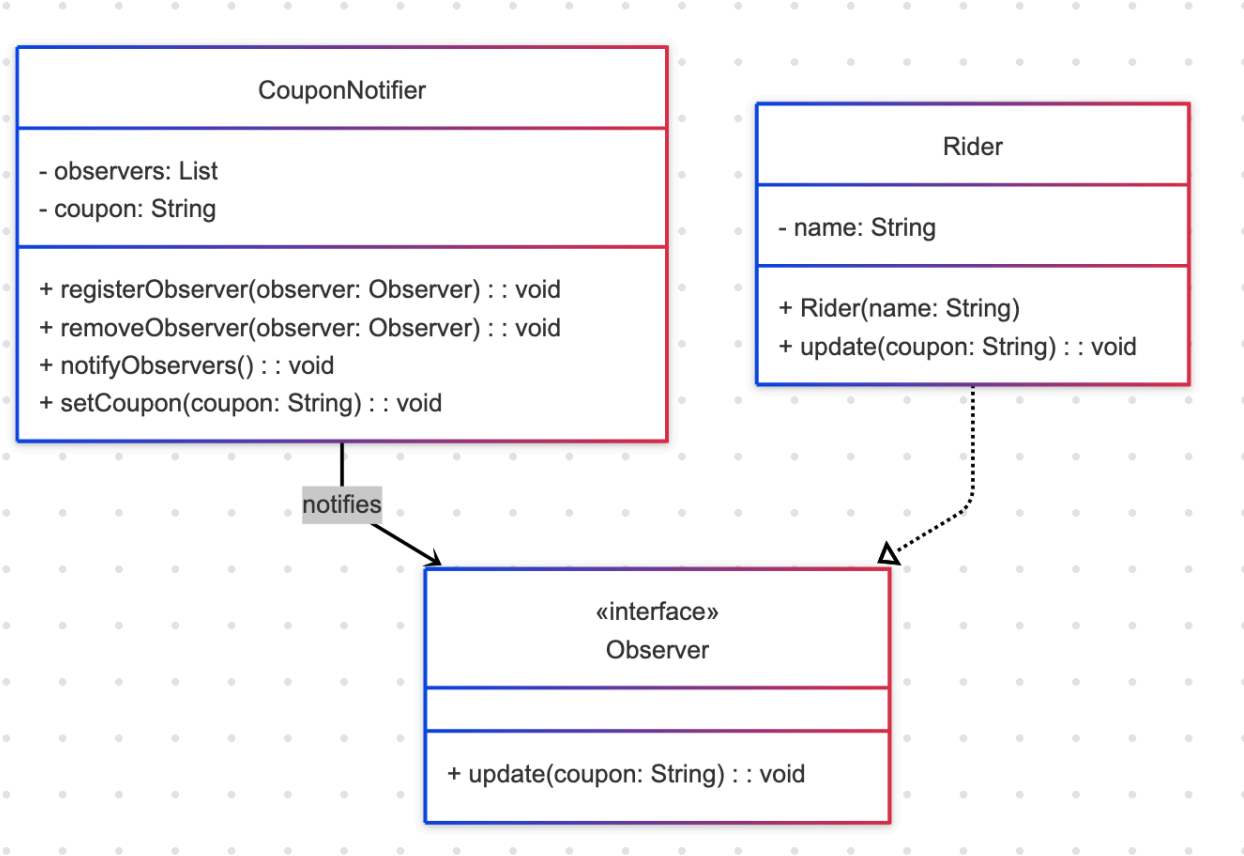
Push Notification: The Observer pattern implements a publish–subscribe mechanism where the subject pushes updates (coupons) to all registered observers (riders) automatically.

Loose Coupling: It decouples the coupon sender from the riders. The sender doesn't need to know the details of the subscribers; it just notifies them.

Dynamic Subscription: Riders can register or unregister at any time, and the notifier will manage the list of subscribers without affecting the rest of the system.

Reusability and Extensibility: Adding new types of observers (for example, riders with different notification preferences) is straightforward without modifying the notifier's core logic.

Justify suggestion + class diagram



Sample code

```
import java.util.ArrayList;

import java.util.List;


// Observer interface defining the update method

public interface Observer {

   void update(String coupon);

}
```

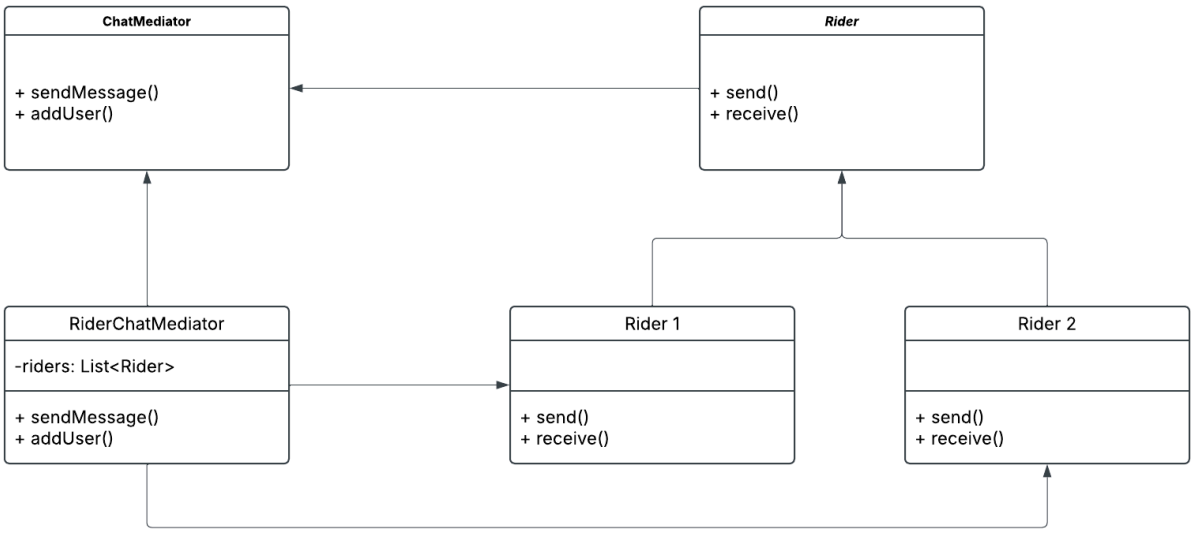
```
// Subject interface (optional in this example, but useful for clarity)
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// CouponNotifier acts as the subject that notifies registered observers
public class CouponNotifier implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String coupon;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
```

```
            observer.update(coupon);
        }
    }

    // When a new coupon is set, notify all observers
    public void setCoupon(String coupon) {
        this.coupon = coupon;
        notifyObservers();
    }
}

// Rider class implements Observer to receive coupon notifications
public class Rider implements Observer {
    private String name;

    public Rider(String name) {
        this.name = name;
    }

    @Override
    public void update(String coupon) {
        System.out.println("Rider " + name + " received coupon: " + coupon);
    }
}
```

```
// Client code to test the coupon push-mode notification system
public class CouponClient {
    public static void main(String[] args) {
        // Create the coupon notifier (subject)
        CouponNotifier notifier = new CouponNotifier();

        // Create riders (observers)
        Rider rider1 = new Rider("Alice");
        Rider rider2 = new Rider("Bob");

        // Register riders to receive coupon updates
        notifier.registerObserver(rider1);
        notifier.registerObserver(rider2);

        // Set a new coupon; all registered riders will be notified automatically
        notifier.setCoupon("DISCOUNT2025");
    }
}
```

Nico [7] 7. You want to let the system serve as a social network for riders. A rider can post a message to other riders, sometimes broadcast to other riders. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Mediator - The Mediator pattern allows for simple and effective facilitation between objects, by creating a mediator object for the rideshare app different users can send each other direct messages or post on a message board all of which will utilize the mediator under the hood.



Sample code:

```
interface ChatMediator {

    void sendMessage(String msg, Rider sender);

    void addUser(Rider user);

}



class RiderChatMediator implements ChatMediator {

    private List<Rider> riders = new ArrayList<>();



    @Override

    public void addUser(Rider rider) {

        this.riders.add(rider);

    }



    @Override

    public void sendMessage(String msg, Rider sender) {
```

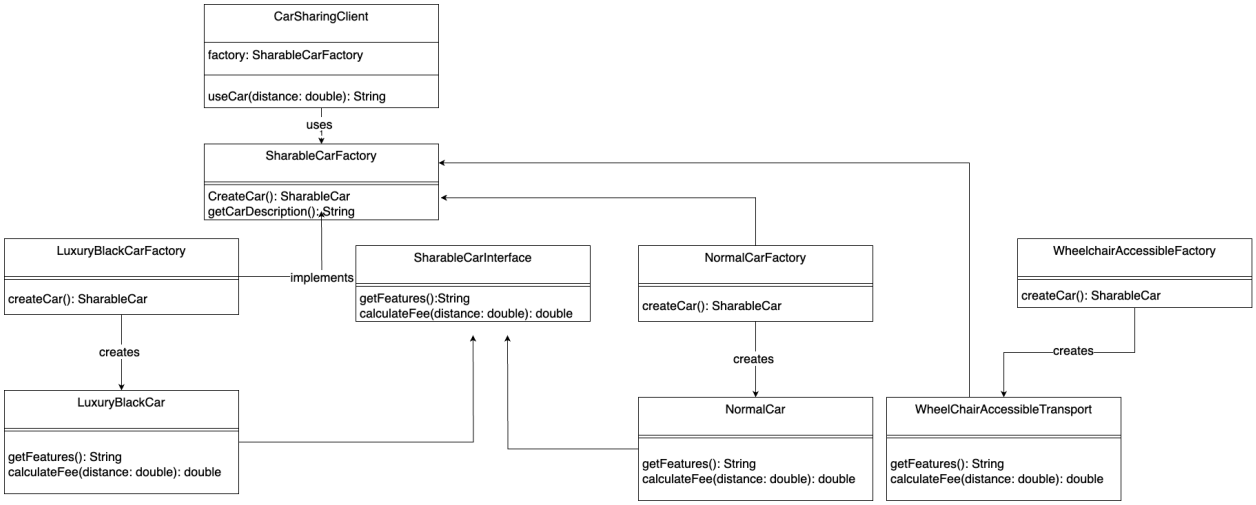
```
        for (Rider rider : riders) {
            // message should not be received by the user sending it
            if (rider != sender) {
                rider.receive(msg);
            }
        }
    }
}

class Rider {
    private String name;
    private ChatMediator mediator;

    public Rider(String name, ChatMediator mediator) {
        this.name = name;
        this.mediator = mediator;
    }

    public void send(String message) {
        System.out.println(this.name + " sends: " + message);
        mediator.sendMessage(message, this);
    }

    public void receive(String message) {
        System.out.println(this.name + " received: " + message);
    }
```

```
}


// Client code
public class Main {
    public static void main(String[] args) {
        ChatMediator mediator = new RiderChatMediator();
        Rider john = new Rider("John", mediator);
        Rider jane = new Rider("Jane", mediator);
        mediator.addUser(john);
        mediator.addUser(jane);


        john.send("Hi there!");
        jane.send("Hey!");
    }
}
```

MAX: [7] 8. You plan to implement a sharableCar object in your system, which could be categorized into normal car, luxury black car, SUV, wheelchair accessible transport, etc. Each category of car may carry and display different features, as well as different transportation fee calculation methods. To keep the code consistent with extensibility, you would like to create such sharableCar object without exposing the creation logic and refer to newly created object using a common interface. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Design pattern: factory pattern

The factory pattern can create different car objects without showing creation logic.



SharableCar.java:

```
package com.example.factory;


public interface SharableCar {
   String getFeatures();
   double calculateFee(double distance);
}
```

SharableCarFactory.java:

```
package com.example.factory;
```

```
public abstract class SharableCarFactory {
  public abstract SharableCar createCar();

  public String getCarDescription() {
    SharableCar car = createCar();
    return "Car Features: " + car.getFeatures() + ", Fee for 10km: $" + car.calculateFee(10.0);
  }
}
```

NormalCar.java:

```
package com.example.factory;

public class NormalCar implements SharableCar {
  @Override
  public String getFeatures() {
    return "Basic seating, standard comfort";
  }

  @Override
  public double calculateFee(double distance) {
    return distance * 1.5; // $1.5 per km
  }
}
```

LuxuryBlackCar.java:

```
package com.example.factory;
```

```
public class LuxuryBlackCar implements SharableCar {
  @Override
  public String getFeatures() {
    return "Leather seats, premium service";
  }

  @Override
  public double calculateFee(double distance) {
    return distance * 3.0; // $3.0 per km
  }
}
```

SUV.java:

```
package com.example.factory;

public class SUV implements SharableCar {
  @Override
  public String getFeatures() {
    return "Spacious, off-road capable";
  }

  @Override
  public double calculateFee(double distance) {
    return distance * 2.0; // $2.0 per km
  }
}
```

WheelchairAccessibleTransport.java:

```
package com.example.factory;

public class WheelchairAccessibleTransport implements SharableCar {
  @Override
  public String getFeatures() {
    return "Wheelchair ramp, extra space";
  }

  @Override
  public double calculateFee(double distance) {
    return distance * 2.5; // $2.5 per km
  }
}
```

NormalCarFactory.java:

```
package com.example.factory;

public class NormalCarFactory extends SharableCarFactory {
  @Override
  public SharableCar createCar() {
    return new NormalCar();
  }
}
```

LuxuryBlackCarFactory.java:

```
package com.example.factory;

public class LuxuryBlackCarFactory extends SharableCarFactory {
```

```
  @Override
  public SharableCar createCar() {
    return new LuxuryBlackCar();
  }
}
```

SUVFactory.java:

```
package com.example.factory;

public class SUVFactory extends SharableCarFactory {
  @Override
  public SharableCar createCar() {
    return new SUV();
  }
}
```

WheelchairAccessibleFactory.java:

```
package com.example.factory;

public class WheelchairAccessibleFactory extends SharableCarFactory {
  @Override
  public SharableCar createCar() {
    return new WheelchairAccessibleTransport();
  }
}
```

CarSharingClient.java:

```
package com.example.factory;

public class CarSharingClient {
```

```
    private final SharableCarFactory factory;

    public CarSharingClient(SharableCarFactory factory) {
        this.factory = factory;
    }

    public String useCar(double distance) {
        SharableCar car = factory.createCar();
        return "Using car: " + car.getFeatures() + ", Fee for " + distance + "km: $" + car.calculateFee(distance);
    }
}
```

FactoryPatternTest.java:

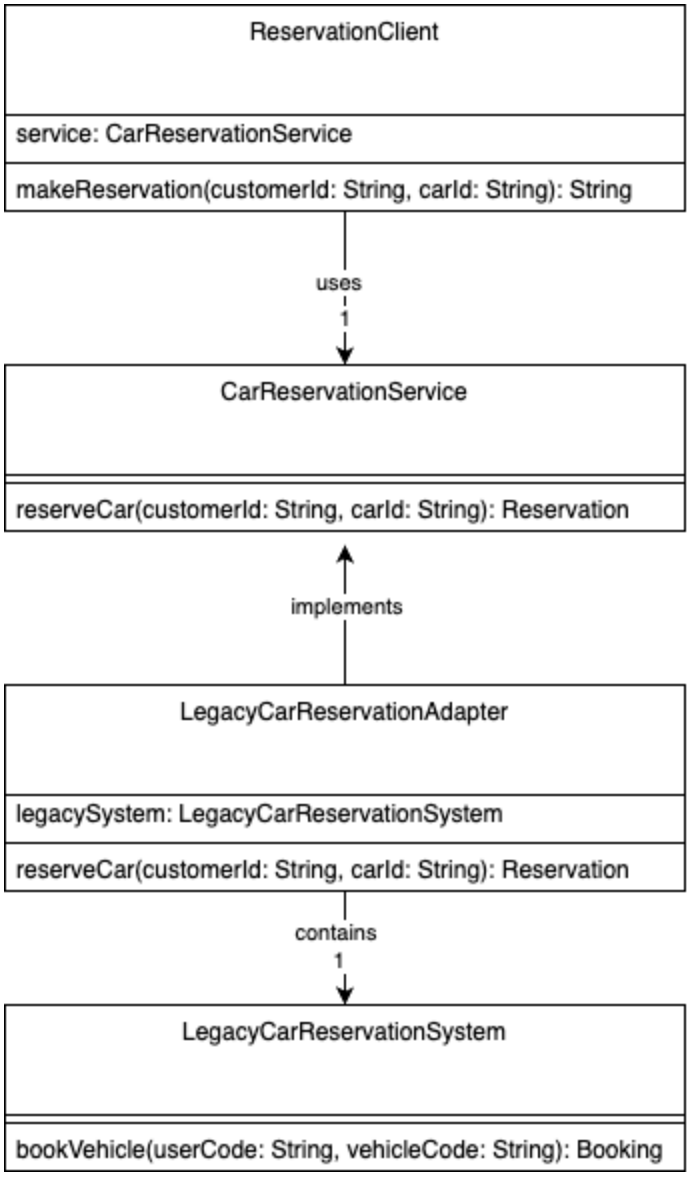
```
package com.example.factory;

public class FactoryPatternTest {
    public static void main(String[] args) {
        // Test with NormalCar
        SharableCarFactory normalFactory = new NormalCarFactory();
        CarSharingClient normalClient = new CarSharingClient(normalFactory);
        System.out.println("Normal Car Test: " + normalClient.useCar(5.0));
        System.out.println(normalFactory.getCarDescription());

        // Test with LuxuryBlackCar
        SharableCarFactory luxuryFactory = new LuxuryBlackCarFactory();
        CarSharingClient luxuryClient = new CarSharingClient(luxuryFactory);
        System.out.println("Luxury Black Car Test: " + luxuryClient.useCar(5.0));
        System.out.println(luxuryFactory.getCarDescription());
```

```
        // Test with SUV
        SharableCarFactory suvFactory = new SUVFactory();
        CarSharingClient suvClient = new CarSharingClient(suvFactory);
        System.out.println("SUV Test: " + suvClient.useCar(5.0));
        System.out.println(suvFactory.getCarDescription());

        // Test with WheelchairAccessibleTransport
        SharableCarFactory wheelchairFactory = new WheelchairAccessibleFactory();
        CarSharingClient wheelchairClient = new CarSharingClient(wheelchairFactory);
        System.out.println("Wheelchair Accessible Test: " + wheelchairClient.useCar(5.0));
        System.out.println(wheelchairFactory.getCarDescription());
    }
}
```

MAX: [7] 9. You plan to leverage some legacy code in your company that implements online car reservation function. That means you want to turn those legacy code as reusable components in your new system but with different interfaces. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Design pattern: Adapter pattern

Justification: The adapter pattern acts as a bridge to make the legacy code work with the new system



Java Code:

CarReservationService.java:

```
package com.example.adapter;
```

```
// The interface the new system expects
public interface CarReservationService {
  String reserveCar(String customerId, String carId);
}
```

LegacyCarReservationSystem.java:

```
package com.example.adapter;

// The legacy system with its own method
public class LegacyCarReservationSystem {
  public String bookVehicle(String userCode, String vehicleCode) {
    return String.format("Booking confirmed for user %s and vehicle %s", userCode, vehicleCode);
  }
}
```

LegacyCarReservationAdapter.java:

```
package com.example.adapter;

// Adapts the legacy system to the new interface
public class LegacyCarReservationAdapter implements CarReservationService {
  private final LegacyCarReservationSystem legacySystem;

  public LegacyCarReservationAdapter() {
    this.legacySystem = new LegacyCarReservationSystem();
  }

  @Override
```

```
    public String reserveCar(String customerId, String carId) {
        return legacySystem.bookVehicle(customerId, carId); // Adapts the call
    }
}
```

ReservationClient.java:

```
package com.example.adapter;

// The client that uses the reservation service
public class ReservationClient {
    private final CarReservationService reservationService;

    public ReservationClient(CarReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public String makeReservation(String customerId, String carId) {
        return reservationService.reserveCar(customerId, carId);
    }
}
```

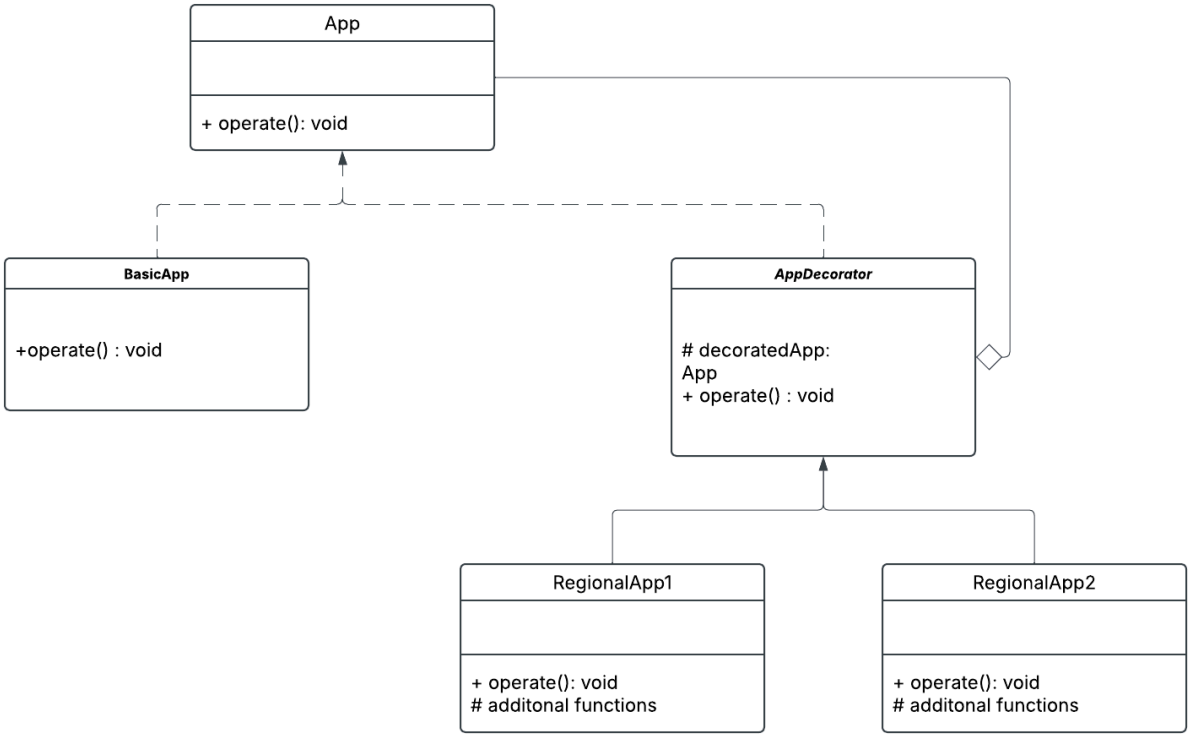
AdapterPatternTest.java:

```
package com.example.adapter;

// Test class to demonstrate the Adapter pattern
public class AdapterPatternTest {
    public static void main(String[] args) {
        // Create the adapter
        CarReservationService adapter = new LegacyCarReservationAdapter();
```

```
        // Create the client with the adapter
        ReservationClient client = new ReservationClient(adapter);

        // Test Case 1: Simple reservation
        String result1 = client.makeReservation("CUST001", "CAR101");
        System.out.println("Test 1: " + result1);

        // Test Case 2: Different customer and car
        String result2 = client.makeReservation("CUST002", "CAR202");
        System.out.println("Test 2: " + result2);

        // Test Case 3: Edge case with null (for robustness)
        try {
            String result3 = client.makeReservation(null, "CAR303");
            System.out.println("Test 3: " + result3);
        } catch (Exception e) {
            System.out.println("Test 3: Failed as expected - " + e.getMessage());
        }
    }
}
```

Nico [7] 10. As you are expanding your services into new cities, you may need to add some new features/options to some services. You don't want to change the code structure (because some cities may need the original version) while you add new functions. Any design pattern you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Decorator - The Decorator pattern allows behaviors to be added to different objects dynamically without impacting the functioning of other objects of the same class. This would be particularly effective in this case since as the app moves into different regions the app in those regions can be “decorated” to match local regulations, price levels, etc. With the decorator framework the app can easily expand into new areas without the need to make changes to the regions where the app is already functioning.



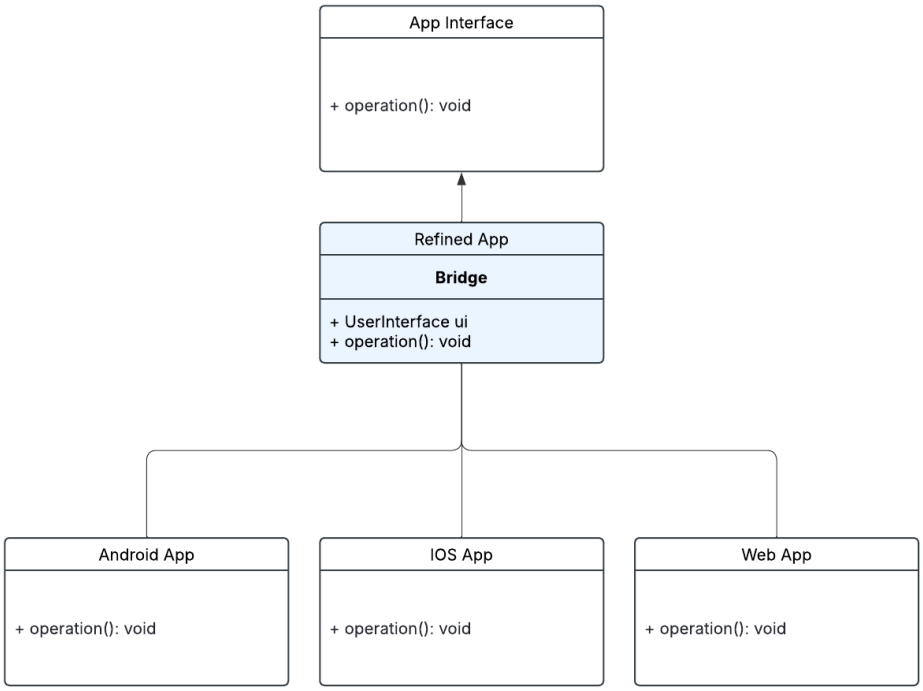

```
interface ServiceFeature {

    void operate();

}



class BasicService implements ServiceFeature {

    public void operate() {

        System.out.println("Performing basic service operations.");

    }

}
```

```
abstract class FeatureDecorator implements ServiceFeature {
    protected ServiceFeature decoratedFeature;

    public FeatureDecorator(ServiceFeature feature) {
        this.decoratedFeature = feature;
    }

    public void operate() {
        decoratedFeature.operate();
    }
}

class NewFeatureDecorator extends FeatureDecorator {
    public NewFeatureDecorator(ServiceFeature feature) {
        super(feature);
    }

    public void operate() {
        super.operate();
        addFeature();
    }

    private void addFeature() {
        System.out.println("Adding new feature.");
    }
```

```
}


// Client code

ServiceFeature service = new BasicService();

ServiceFeature decoratedService = new NewFeatureDecorator(service);

decoratedService.operate();
```

Nico [7] 12. To be practical, your software will require the drivers to have a smartphone and the users must have access to either a smartphone or the mobile website. While drivers and users may use different types of devices to view your app, which architectural style/design pattern would you see appropriate here? Briefly justify your suggestion, and draw a class diagram to explain your design (maybe with short descriptions if appropriate). For the design pattern you recommend to apply, work out some sample code (make sure to have client code to test).

Bridge - The Bridge pattern allows separation between the underlying solution and the implementation of that solution on disparate devices this allows for device specific modifications to implement the solution (on android for example) that do not disrupt the functionality of the solution as a whole.



.

Sample code:

```
interface AppInterface {
    void operation();
}

class IOSApp implements AppInterface {
    private UserInterface userInterface;

    public IOSApp(UserInterface ui) {
        this.userInterface = ui;
    }

    public void operation() {
        userInterface.interact();
    }
}

interface UserInterface {
    void interact();
}

class SmartphoneUserInterface implements UserInterface {
```

```
    public void interact() {
        System.out.println("Interaction through smartphone app.");
    }
}


class WebApp implements AppInterface {
    private UserInterface userInterface;

    public WebApp(UserInterface ui) {
        this.userInterface = ui;
    }

    public void operation() {
        userInterface.interact();
    }
}

interface UserInterface {
    void interact();
}

class WebUserInterface implements UserInterface {
    public void interact() {
        System.out.println("Interaction through smartphone app.");
    }
}
//extend with android app etc.

// Client code
UserInterface uiPhone = new SmartphoneUserInterface();
AppInterface PhoneApp = new IOSApp(uiPhone);
UserInterface uiWeb = new WebUserInterface();
AppInterface webAppEx = new WebApp(uiWeb);
PhoneApp.operation();
webAppEx.operation();
```