ChatGPT

Ride-Sharing System

A comprehensive C++ implementation of a ride-sharing service that connects riders with drivers, manages ride requests, and handles payments and ratings.

How to Run This Project

```
# Step 1: Compile the project
# Ensure you have g++ (C++17) installed on your system

# Compile all the necessary source files
# (Assumes your project structure is as follows: models/, strategy/, pricing/, manager/)
g++ -std=c++17 -I. main.cpp models/*.cpp strategy/*.cpp pricing/*.cpp manager/*.cpp -o ride_sharing

# Step 2: Run the executable
./ride_sharing
```

Table of Contents

System Overview

The ride-sharing system is designed to facilitate the following core operations: - Rider and driver registration - Ride booking and driver matching - Dynamic pricing based on various factors - Real-time location tracking - Rating system for both riders and drivers - Ride status management throughout the journey

Functional Requirements

1. **User Management**
2. Register riders and drivers with personal details

3. Maintain user profiles with ratings
4. Track driver availability status
5. Store user contact information
6. **Vehicle Management**
7. Support multiple vehicle types (BIKE, CAR, AUTO, SUV)
8. Store vehicle details (registration, model, color)
9. Associate vehicles with drivers
10. **Ride Management**
11. Create ride requests with pickup/drop locations
12. Match riders with nearby drivers
13. Track ride status (CREATED, ACCEPTED, STARTED, COMPLETED)
14. Calculate ride fare dynamically
15. Process ride completion and payments
16. **Location Services**
17. Track real-time location of drivers
18. Calculate distance between locations
19. Find nearest available drivers
20. **Rating System**
21. Allow riders to rate drivers
22. Allow drivers to rate riders
23. Maintain rating history
24. Calculate average ratings

Non-Functional Requirements

1. **Performance**
2. Fast driver matching algorithm
3. Efficient location-based search
4. Quick fare calculation
5. **Scalability**
6. Support for multiple concurrent rides
7. Easily add new vehicle types

8. Extensible pricing strategies

9. **Maintainability**

10. Modular code structure
11. Clear separation of concerns

12. Well-documented classes and methods

13. **Reliability**

14. Proper error handling
15. Data validation

16. System state consistency

17. **Security**

18. Protected user data
19. Secure payment processing
20. Access control for sensitive operations

Design Patterns Used

1. **Strategy Pattern**
2. `MatchingStrategy`: For implementing different driver matching algorithms
3. `PricingStrategy`: For implementing various pricing models

4. Allows runtime selection of algorithms

5. **Factory Pattern**

6. Used for creating different types of vehicles

7. Ensures proper initialization of complex objects

8. **Observer Pattern**

9. For ride status updates

10. Notification system implementation

11. **Singleton Pattern**

12. RideManager implementation
13. Ensures single point of control for ride operations

SOLID Principles Implementation

1. **Single Responsibility Principle (SRP)**
2. Each class has a single, well-defined purpose
3. Example: `Location` class handles only location-related operations
4. **Open/Closed Principle (OCP)**
5. New strategies can be added without modifying existing code
6. Vehicle types can be extended without changing core logic
7. **Liskov Substitution Principle (LSP)**
8. `Driver` and `Rider` properly inherit from `User`
9. All strategy implementations are substitutable
10. **Interface Segregation Principle (ISP)**
11. Separate interfaces for different strategies
12. Clean separation between user types
13. **Dependency Inversion Principle (DIP)**
14. High-level modules depend on abstractions
15. Strategy pattern implementations follow DIP

Core Components

1. **Models**
2. `User`: Base class for system users
3. `Driver`: Manages driver-specific attributes
4. `Rider`: Handles rider-specific functionality
5. `Vehicle`: Stores vehicle information
6. `Location`: Handles geographical coordinates
7. `Ride`: Manages ride lifecycle
8. `RideRequest`: Encapsulates ride request details
9. **Strategies**
10. `MatchingStrategy`: Interface for driver matching
11. `NearestDriverStrategy`: Implementation of driver matching
12. `PricingStrategy`: Interface for price calculation

13. `DynamicPricingStrategy`: Implementation of dynamic pricing

14. **Managers**

15. `RideManager`: Orchestrates ride operations
16. Handles driver-rider matching
17. Manages ride lifecycle

Building and Running

```
# Compile the project
g++ -std=c++17 -I. main.cpp models/*.cpp strategy/*.cpp pricing/*.cpp manager/*.cpp -o ride_sharing

# Run the executable
./ride_sharing
```

Sample Output

```
Initial ride details:
Ride ID: RIDE_REQ1
Status: 2

Driver Details:
Name: Sham kumar
ID: D1
Phone: 9876543210
Email: Sham@example.com
Current Location: (12.9716, 77.5946)

Vehicle Details:
Vehicle ID: V1
Registration Number: KA01AB1234
Type: Car
Model: Toyota Camry
Color: White

Rider Details:
Name: Ram Kunar
ID: R1
Phone: 1234567890
Email: Ram@example.com

Ride Information:
Pickup Location: (12.9717, 77.5947)
```

```
Drop Location: (12.9718, 77.5948)
Estimated Price: $0.186312
Type: Car
Model: Toyota Camry
Color: White

Rider Details:
Name: Ram Kunar
ID: R1
Phone: 1234567890
Email: Ram@example.com

Ride Information:
Pickup Location: (12.9717, 77.5947)
Drop Location: (12.9718, 77.5948)
Estimated Price: $0.186312

Ride completion details:
"Completing ride..."
Updated ride status: 5
Final price: $0.186312

Trip Details:
Total Distance: 0.02 km

Rating information:
"Rating driver and rider..."
Final Ratings:
Driver Rating: 4.85
Rider Rating: 4.80
```