LLD Hackathon Challenge: Design a Ride-Sharing Service Platform

Background

Imagine a new ride-sharing startup (think of a simplified Uber/Ola - **Give any hypothetical name to your application**) that needs a robust backend system designed from scratch. As a good developer who has mastered SOLID principles and design patterns, you are tasked with designing an **in-memory, object-oriented ride-sharing service**. You will design the system's architecture, create UML diagrams, and implement a fully working prototype (in-memory code only) that demonstrates the core functionality. The goal is to showcase clean **Low-Level Design (LLD)** using multiple design patterns (Strategy, Factory, Singleton, Observer, Decorator) and SOLID principles.

This challenge is open-ended enough to allow creativity in your approach, while ensuring that key requirements are met. We're looking for a balance of correct functionality, code quality, and design elegance.

Problem Statement

Design an **end-to-end ride-sharing service platform**.

The system should enable riders to book rides and get matched with drivers in an imaginary city. You must define the architecture and class design, draw relevant UML diagrams, and implement the core logic in code (running in a single-threaded environment, no external frameworks or databases). The design should be extensible and maintainable, leveraging the design patterns and principles you've learned. Focus on clean **object-oriented design** rather than UI – there is no need for a web/mobile interface, just demonstrate the backend logic via classes and methods.

Your solution should cover everything from managing users and rides to the dispatch/matching process and ride lifecycle (from booking to completion). Additionally, you should address how the system handles dynamic behaviors like notifications and pricing in a flexible way. The problem expects you to apply multiple design patterns appropriately in different parts of the system, and clearly document your design decisions.

Functional Requirements:

1. **User Management:** Ability to register and manage two types of users – **Riders** and **Drivers**.
    - Riders have basic details (e.g. name, phone) and maybe a current location or default pickup point.
    - Drivers have details like name, vehicle info, and current location/status (available or on-trip).
    - *Hint:* Assume authentication is trivial & handled elsewhere (focus on design, not login flow).
2. **Ride Booking:** A Rider can request a ride by providing a pickup location and a drop-off location.
    - When a ride request is made, the system should create a **Ride** object capturing details (rider, pickup/drop locations, requested vehicle type, etc.).
    - The system then needs to assign an appropriate Driver to this ride request (see Driver Matching below).
3. **Driver Matching Algorithm :** The system should match a ride request to an available Driver efficiently.
    - For simplicity, assume a single city or a limited area. You can represent locations in a simple way (e.g., coordinates or region names) without external map APIs.
    - Implement 2 matching algorithms as of now. The "**nearest**" driver and the "**best driver based on rating**". This selection logic **should be easily changeable**, as the company might experiment with different matching algorithms.
    - If a driver is not available or rejects the ride, the system should try the next best driver (if any). You can simulate a simple accept/reject or assume drivers always accept this prototype, but design for the possibility

of rejection.

4. **Ride Workflow:** Once a Driver is assigned and accepts:
    - Mark the ride as **confirmed** and notify the Rider and Driver that the ride is starting (e.g., “Driver on the way”).
    - Simulate the ride status progression: e.g., “Driver en route to pickup”, “Ride in progress (passenger onboard)”, and “Ride completed”. You can simulate the Driver moving from pickup to drop-off in steps. The system should allow tracking the ride status.
    - The Rider should be able to get updates about the Driver’s status or location during the trip (for example, by checking the ride status or through a notification mechanism).
5. **Vehicle/Ride Types:** You should support multiple ride types (e.g., “Bike”, “Sedan”, “SUV”, “Auto-Rickshaw”). Each type might have different attributes (capacity, fare rates) and possibly slightly different handling. Also, your Application should allow normal rides and carpooling rides.
6. **Ride Completion and Payment:** When the ride is completed:
    - Calculate the fare for the ride. Start with a simple fare calculation (say, a base fare plus distance-based cost).
    - The design should allow **additional pricing rules** to be applied (for example: surge pricing multiplier at peak hours, or discounts/promo codes). These additional rules might not be implemented in full due to time, but your design should make it easy to plug them in if needed.
    - Mark the ride as finished, and record the transaction (payment can be simulated as always successful). For this challenge, you don’t need to integrate actual payment gateways – just assume payment is done and perhaps output the fare.
7. **Notifications:** The system should notify relevant parties of important events.
    - When a driver is assigned or the ride status changes, the Rider should be notified (and potentially the Driver app as well).

    - When payment is completed, send a confirmation to the Rider.
    - Assume any other types of notifications required as per your design.
    - These notifications can be simulated with simple print statements in the code, but the design should support sending notifications without tightly coupling the notification logic to the core ride logic.

8. **In-Memory Data Management:** All data (users, rides, etc.) should be stored in memory using appropriate data structures.
    - You should have a way to keep track of currently available drivers, ongoing rides, and maybe a history of completed rides (if needed for demonstration).
    - Ensure that adding or removing drivers (e.g., driver goes offline or comes online) is reflected in the available pool.
    - No need to connect to any DB. Keep everything in Data-structures.
9. **Single-Threaded Operation:** Assume all actions occur sequentially (one at a time) in a single process. Concurrency is **not** required. For example, you might simulate multiple ride requests one after another in a main program or test routine to demonstrate functionality.

(Feel free to make reasonable assumptions to simplify the domain as long as you state them. For example, you can assume all rides are in one city and distances can be hardcoded or calculated via a simple function since mapping isn't the focus.)

Non-Functional Requirements:

- **SOLID Design:** The solution **must adhere to SOLID principles**:
    - Each class should have a clear Single Responsibility.
    - The system should be Open for extension (e.g., easy to add new ride types or new matching strategies) but Closed for modification (new features shouldn't require rewriting core logic).
    - Use abstractions (interfaces or abstract classes) so that components are substitutable (Liskov Substitution) and high-level logic isn't tightly

  coupled to low-level implementations (Dependency Inversion).
  - Use Interface Segregation to keep class and interface definitions focused.

- **Use of Design Patterns:** Apply the design patterns covered in the playlist where appropriate. In particular:
  - **Strategy:** Use it to keep certain algorithms or logic pluggable.
  - **Factory:** Use factories for object creation where needed.
  - **Singleton:** Likely needed for manager classes that orchestrate core functions (e.g., a single Dispatch or RideManager that holds the list of drivers and rides).
  - **Observer:** Employ observers to decouple event notifications from the main logic. For example, when a ride's status changes, observers (like a Rider's app interface or a logging service) can be notified without the Ride service knowing about all of them.
  - **Decorator:** Design the system such that additional behaviors can be attached to core components dynamically. For example, a fare calculation might be decorated with a surge pricing component or a discount component.
- **Design for Extensibility:** Think of some features that are **not** required now but could be in the future: scheduling rides in advance, driver ratings and reviews, etc. **You don't need to implement these**, but if your design can accommodate them with minimal changes, that's a sign of a good design. For instance, how would you add a "schedule ride for later" feature?
- **Clarity and Modularity:** The code and class design should be clear to read and logically organized. Someone new should be able to understand the high-level architecture from your documentation and navigate through the codebase easily. Use meaningful naming conventions and comments/documentation where helpful.
- **Pluggable Matching Logic:** Design the driver selection (matching) as an interchangeable module. Today it might be "find nearest driver", tomorrow it might be "find highest-rated driver" or a combination.

- **Performance (within scope):** Within a single-process simulation, the system should handle a reasonable number of objects (say hundreds of drivers/rides) efficiently. Use appropriate data structures for lookups (e.g., if matching the nearest driver by location, how will you store/query drivers?).

  Note: You do *not* need to design for million-scale concurrency for this hackathon; focus on correct and clean behavior, but don't use naive algorithms that are unnecessarily slow for the scale of our simulation.

- **No External Libraries/Frameworks:** You must implement the solution using core language features (use any mainstream OOP language – Java, C++, Python, etc. – but without heavy frameworks). Do **not** use databases or web servers; simulate all data and actions in-memory. This keeps the hackathon centric to Low level design only.

- **Single-Thread Assumption:** Since the system is single-threaded, you don't need to handle thread safety or concurrency issues. However, design your code in a way that if extended to multi-threaded in the future, the core logic could still hold. This is more of a consideration for best practices – you will not be tested on multi-threading here.

Constraints and Assumptions

- **Time Constraint:** You have a complete **1 Week** to submit your Project. This implies you should prioritize core functionality and good design over edge-case completeness. It's okay if some minor features are mocked or simplified, as long as the design supports extending them.

- **Trade-offs:** You are allowed to take trade-offs in SOLID principles as they are not hard rules and all real world applications are built on trade-offs. You should just be able to defend your trade-off in the application (Why did you take it ? And, how does it help you ?)

- **In-Memory Simulation:** As stated, everything is in-memory. You can pre-load some data for demo purposes (e.g., a list of drivers with preset locations, to demonstrate matching). Feel free to simulate passage of time or movement by using loops or incremental updates in your code.

- **Input/Output:** There's no need to build a fancy input/output interface. You can demonstrate functionality via a simple text-based sequence (for example, a `main()` method or test function that creates objects and calls methods to simulate a scenario: a rider requests a ride, driver gets assigned, etc., printing out status updates to the console).

- **No Uncovered Topics:** You should not use any concepts beyond the scope of what you've learned. For example, do not introduce multithreading, microservices, or external databases. The focus is on applying **OO design patterns** and principles in a contained environment. If you think a certain complex component is needed (say, real maps or concurrency), you are free to **mock or assume** those parts, and concentrate on the structural design.

Note : There is all the room for creativity. You don't have to use every pattern if it doesn't fit your approach, but the top solutions will likely incorporate multiple patterns effectively. Conversely, avoid using patterns just for the sake of it – justify their usage.

Submission Expectations

To be eligible for the prize and to have your solution evaluated thoroughly, please prepare the following deliverables:

- **Architecture Diagrams:** Clear UML diagrams of your design:
    - *Class Diagram* showing proper classes, interfaces, and relationships in your system. Make sure to include the main entities (Rider, Driver, Ride, etc.), managers/singletons, and any significant patterns (e.g., strategy interfaces, observer relationships, decorators, factories).
    - *Sequence Diagram:* Illustrate one core flow, such as a Rider requesting a ride and the system matching a driver and completing the ride. This should show the interaction between objects over that process (e.g., Rider -> RideManager -> MatchingStrategy -> Driver, etc., along with notifications/events).
- **Code Implementation:** A complete, working **in-memory prototype** of the ride-sharing service:
    - Organize your code logically into classes (and files/modules as appropriate).
    - The code should be able to simulate a few happy flows. For instance, you might create a few drivers and one rider in a `main()` method, then simulate the rider requesting a ride and demonstrate through console

output that the driver was matched, the ride went through various stages, and completed with a fare. Make sure to show that your design works for the basic happy-path scenario (and if possible, also handle an edge case like “no drivers available” gracefully).

  - You do **not** need a fancy UI – just running the code should trigger the simulation. Ensure it’s easy to run (provide instructions if needed, e.g., “compile and run Main.java” or “python main.py”).

- **README / Design Explanation (text file) :** A brief write-up (can be in the README or a separate document) explaining your design choices.
  - Describe how you applied SOLID principles (e.g., which classes follow Single Responsibility, how you ensured Open/Closed, etc.).
  - List the design patterns you used and **where/how** you used them in the system.
  - If any requirements were simplified or assumptions made, trade-offs taken, state them clearly here. This helps us understand the scope of your solution.

Submit a .zip File for your entire module containing your (Code + UML + ReadMe File), structured properly over CoderArmy App.

The best solutions will balance **correctness, code quality, and adherence to design principles**.

We want to see that you can take a real-world problem, break it down into clean abstractions, and build it in a way that can grow and adapt.

Good luck, and have fun designing your ride-sharing system!

Coder Army