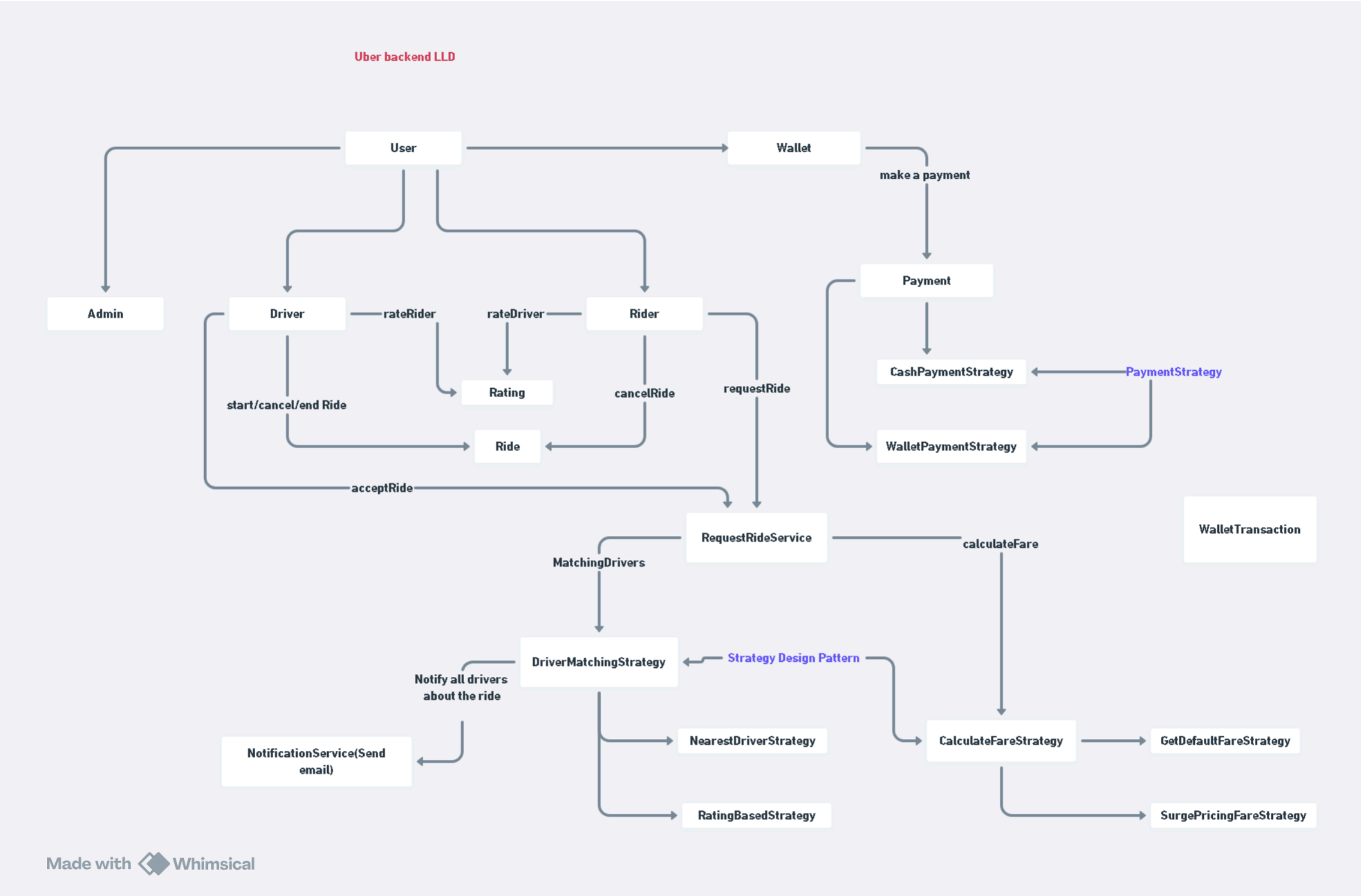
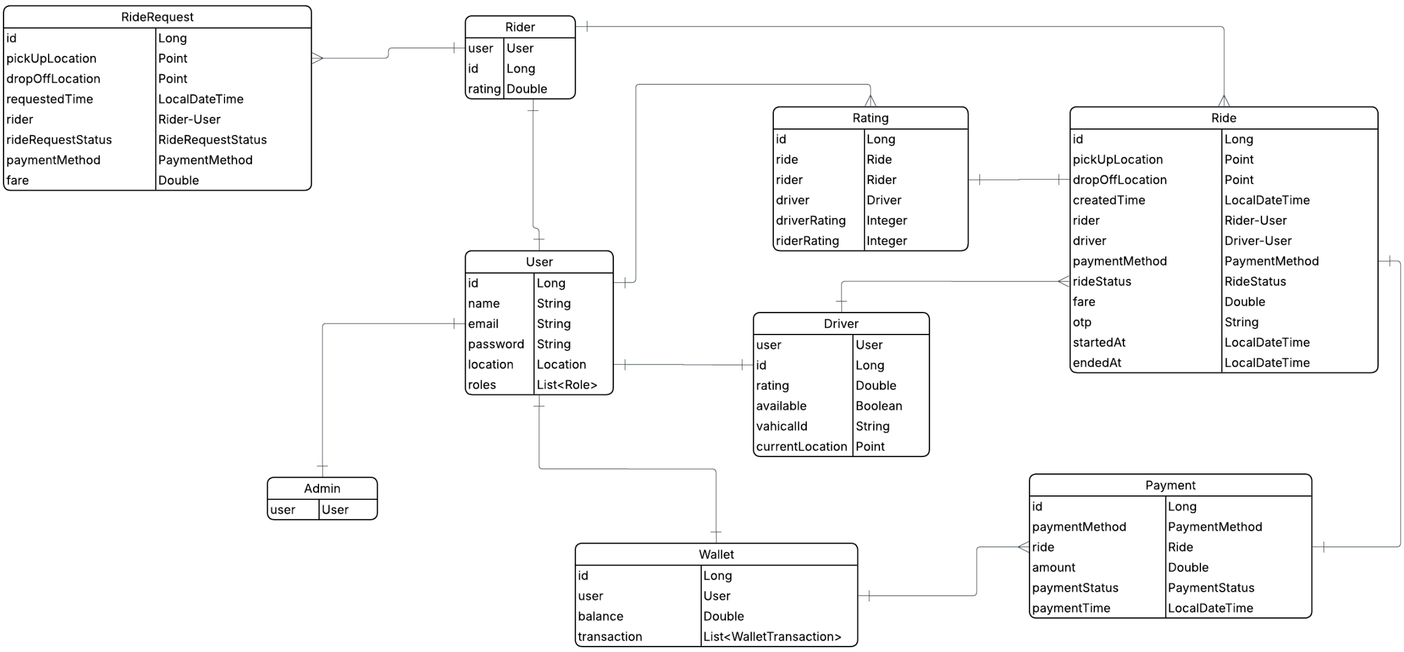
Day 3





1. LLd Diagram

✅ Strong points

- **Single source of identity** (User) with role-specific profiles (Driver, Rider, Admin) — reduces duplication and simplifies auth/RBAC.
- **RideRequestService as orchestrator** — good place to coordinate fare calculation, matching, persistence, and notification.
- **Two independent Strategy families**:
  - CalculateFareStrategy → DefaultFare, SurgePricing
  - DriverMatchingStrategy → NearestDriver, RatingBased

    These are orthogonal and composable — correct.
- **PaymentStrategy abstraction** and Wallet integration — supports multiple payment flows.
- **NotificationService decoupled** — good for async delivery and retries.
- **Rating and Ride entities separate** — keeps behavior/data modeled clearly.

⚠️ Risks, gaps & recommended mitigations

1. **Strategy selection explosion / brittle if-else**
   - *Risk:* Hardcoded if/else in RideRequestService will become unmanageable.
   - *Mitigation:* Use a registry/factory or Spring Map<String, Strategy> injection and encapsulate selection logic in a StrategySelector or small rule engine.
2. **Driver notification & timeouts (async)**
   - *Risk:* Synchronous notify -> blocking waits and bad UX if no driver responds.
   - *Mitigation:* Publish RideRequestedEvent to message bus; drivers respond via events; implement matchmaking window and fallback strategy.
3. **Transaction & persistence ordering**
   - *Risk:* Not persisting the RideRequest before notifying can lead to lost-state on failure.
   - *Mitigation:* Persist request (state = PENDING), publish event, then await driver acceptance. Use event-sourcing/CQRS if you need strong auditability.
4. **Concurrent accept / race conditions**
   - *Risk:* Multiple drivers accept the same request.
   - *Mitigation:* Use optimistic locking / single-winner atomic update (DB row lock / compare-and-swap) or a reservation service that atomically marks RideRequest as ASSIGNED.
5. **Payment flow consistency & failure handling**
   - *Risk:* Payment failure after driver accepted or after ride end.
   - *Mitigation:* Use a saga pattern: steps = reserve fare -> assign driver -> complete ride -> capture payment -> settle. Implement compensation flows for failures.
6. **Scoring/composite matching**
   - *Opportunity:* Combine nearest + rating by weighting rather than mutually exclusive strategies.
   - *Mitigation:* Implement CompositeDriverMatchingStrategy or a scoring pipeline that calculates a score = α*(distance) + β*(rating).
7. **Geo queries & scaling**
   - *Risk:* Naive proximity queries will not scale.

- *Mitigation:* Use spatial indexes (PostGIS, Elasticsearch with geo, Redis geo) and prefilter candidate drivers.

8. **Observability & SLOs**
   - *Recommendation:* Emit metrics for strategy chosen, match latency, failure rates; trace with distributed tracing (OpenTelemetry).
9. **Security & privacy**
   - *Recommendation:* Hash passwords, mask PII in logs, secure admin endpoints, rate-limit sensitive operations.
10. **Testing**
    - *Recommendation:* Unit test each strategy, contract test orchestration, simulate no-driver scenarios and payment failures.

Practical wiring example (Spring-style Strategy registry)

```
// Bean implementations annotated with @Component("nearest") and @Component("ratin
@Service
public class RideRequestService {
    private final Map<String, DriverMatchingStrategy> driverStrategies;
    private final Map<String, CalculateFareStrategy> fareStrategies;

    public RideRequestService(
        Map<String, DriverMatchingStrategy> driverStrategies,
        Map<String, CalculateFareStrategy> fareStrategies) {
        this.driverStrategies = driverStrategies;
        this.fareStrategies = fareStrategies;
    }

    public RideResponse requestRide(RideRequest req) {
        String matchKey = StrategySelector.pickMatchKey(req);
        var strategy = driverStrategies.getOrDefault(matchKey, driverStrategies.get("nearest"))
        List<Driver> candidates = strategy.matchDrivers(req);

        String fareKey = StrategySelector.pickFareKey(req);
        Money fare = fareStrategies.get(fareKey).calculate(req);

        // persist request -> publish event -> return response
    }
```

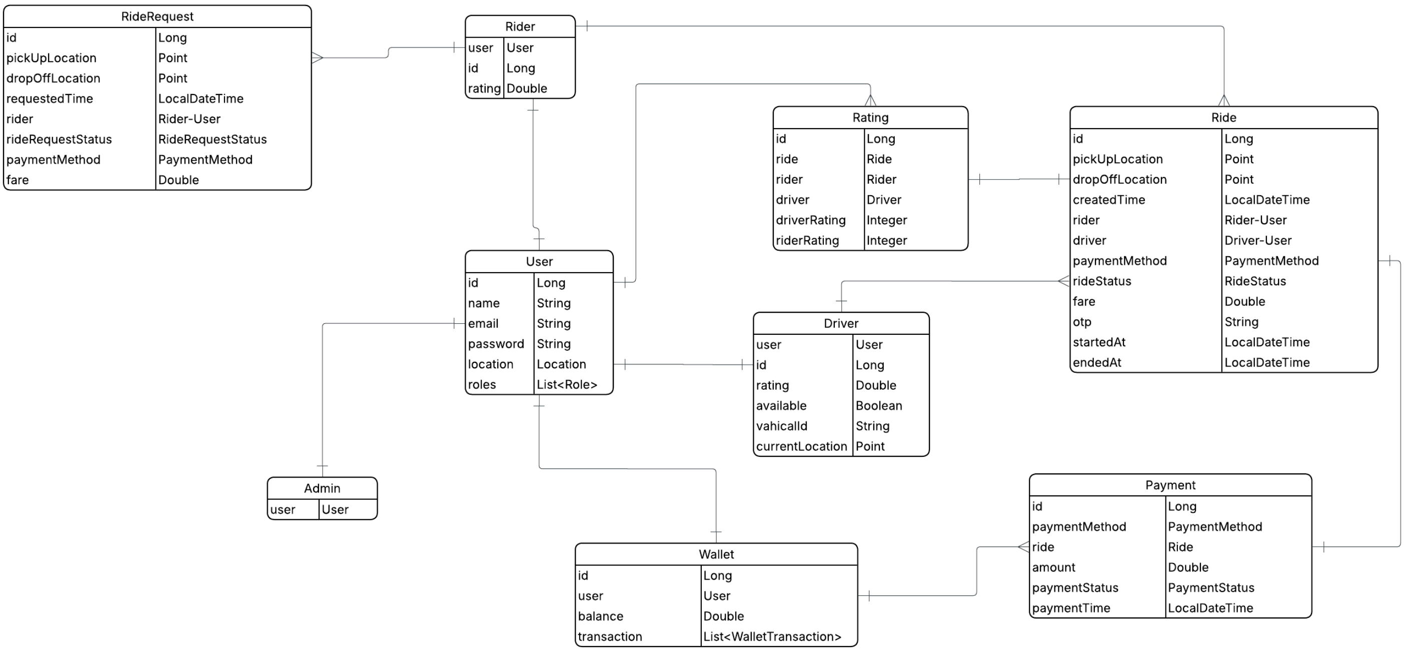
Implementation checklist (ready-to-run)

- Persist RideRequest early (state machine: PENDING → ASSIGNED → ONGOING → COMPLETED → SETTLED).
- Use message bus (Kafka/Rabbit) for RideRequestedEvent and DriverAcceptedEvent.
- Implement atomic assignment (DB lock / reservation service).
- Use strategy registry + StrategySelector for pluggable selection.
- Add composite scoring option for matching.
- Implement payment saga and compensation.
- Add metrics/tracing and alerting on match latency / payment failures.
- Add integration tests for end-to-end happy/failure paths.

Final recommendation (next-highest-impact steps)

1. Wire the strategy registry in code (use the snippet above).
2. Implement the async event flow for notifications and a simple reservation mechanism (atomic assign).
3. Add a minimal saga for payments (reserve/capture) and test failure paths.

2. Uml Diagram



1 — Entity-by-entity explanation

User

Purpose: canonical identity for everyone in the system (rider/driver/admin).
Fields: id, name, email, password, location, roles: List<Role>

Notes: authentication and primary PII live here. Keep this table normalized and small. roles controls RBAC; do not duplicate name/email/password in child tables.

Rider

Purpose: role-profile for users who request rides.
Fields: user: User (1:1), id, rating (double)
Notes: Rider-specific data only; ride history lives in Ride/RideRequest tables. Rating here can be cached/aggregate for quick reads.

Driver

Purpose: role-profile for users who accept and complete rides.
Fields: user: User (1:1), id, rating (double), available (boolean), vehicleId (string), currentLocation (Point)
Notes: currentLocation and available form the basis for proximity queries and matching. Use spatial indexes (PostGIS/Elasticsearch/Redis Geo) for scale.

RideRequest

Purpose: ephemeral request created when a rider asks for a ride (before assignment).
Fields: id, pickUpLocation: Point, dropOffLocation: Point, requestedTime: LocalDateTime, rider: Rider (or User reference), rideRequestStatus: enum (PENDING/EXPIRED/CANCELLED/ASSIGNED), paymentMethod: enum, fare: Double
Notes: Persist early (PENDING) to enable recovery. This is the unit you publish as RideRequestedEvent to the matching system.

DriverMatchingStrategy (conceptual)

Purpose: not a DB entity — design-time pattern. Your service will pick a concrete strategy (nearest / rating-based / composite) to return candidate drivers for a RideRequest.

Ride

Purpose: actual ongoing or completed ride (after driver accepted).
Fields: id, pickUpLocation, dropOffLocation, createdTime, rider, driver, paymentMethod, rideStatus: enum (ASSIGNED/ONGOING/COMPLETED/CANCELLED), fare, otp, startedAt, endedAt
Notes: Ride replaces RideRequest once accepted; it captures lifecycle timestamps and final fare. Keep rideStatus as the state machine source of truth.

Payment

Purpose: records financial transactions for a ride.
Fields: id, paymentMethod: enum, ride: Ride, amount: Double, paymentStatus: enum (PENDING/FAILED/COMPLETED), paymentTime

Notes: Implement as part of a Saga: reserve/authorize → capture → settle. For wallet-based payments, link to Wallet and WalletTransaction.

Wallet & WalletTransaction

Purpose: optional user wallet for in-app balance.
Fields (Wallet): id, user: User, balance: Double, transaction: List<WalletTransaction>
Fields (WalletTransaction): transaction id, amount, type (CREDIT/DEBIT), balanceAfter, createdAt, description, relatedRideId
Notes: Always persist transactions (immutable ledger entries) and update balance in a transactionally-safe manner.

Rating

Purpose: stores ratings after ride completion.
Fields: id, ride: Ride (1:1), rider: Rider, driver: Driver, driverRating: Integer, riderRating: Integer
Notes: Ride → Rating association is useful for audit and recalculation of aggregate ratings on Rider/Driver.

Admin

Purpose: operational user with permissions.
Fields: user: User (1:1), permissions: Set<AdminPermission>, createdAt/updatedAt
Notes: Admins should be minimal and hold only policy/permission metadata.

2 — Relationships & cardinalities (summary)

- User 1 — 1 Rider / 1 — 1 Driver / 1 — 1 Admin (optional role profiles)
- Rider 1 — * RideRequest (a rider can create many requests)
- RideRequest 1 — 1 Ride (when a request is accepted/converted)
- Ride 1 — 1 Payment (usually), and 1 — 1 Rating (post-ride)
- Wallet 1 — 1 User ; Wallet 1 — * WalletTransaction

Use foreign keys for referential integrity and controlled cascading rules (e.g., DO NOT cascade delete rides when user deleted — instead soft-delete or archive).

3 — Typical runtime flow (entire sequence)

1. **Create** RideRequest (state = PENDING) — persist immediately.
2. **Calculate fare** using CalculateFareStrategy (Default/Surge) → set fare on request.
3. **Match drivers** using chosen DriverMatchingStrategy → produce candidate driver list.
4. **Notify drivers** (publish RideRequestedEvent on message bus). Drivers receive push / app notification.
5. **Driver accepts** → atomic assignment (compare-and-swap / DB row lock / reservation service). On success: create Ride from RideRequest, mark RideRequest ASSIGNED.

6. **Start ride** → driver marks startedAt, change rideStatus = ONGOING.
7. **End ride** → driver marks endedAt, rideStatus = COMPLETED. Final fare may be adjusted.
8. **Payment**: Start payment saga — if paymentMethod == WALLET, debit wallet (create WalletTransaction, update Wallet.balance), else call external provider to charge. Mark Payment accordingly.
9. **Rating**: Rider and driver submit ratings → persist Rating and update aggregate ratings on Driver and Rider.
10. **Post-processing**: receipts, invoices, dispute resolution, logs.

4 — Important operational & data concerns

Persistence & Ordering

- Persist RideRequest before notifying drivers. Use DB transactions for consistent state transitions.
- Consider storing the request and publishing an event in the same **transactional outbox** pattern to avoid lost events.

Concurrency (very important)

- **Multiple drivers accepting**: resolve by atomic update on Ride/RideRequest. Use optimistic locking (@Version) or a small reservation service that atomically assigns the ride to a driver.
- **Wallet race conditions**: update balance and write WalletTransaction inside the same DB transaction with proper isolation.

Indexes and geo queries

- Add **spatial index** on Driver.currentLocation and RideRequest.pickUpLocation (GIST/PostGIS or Elasticsearch geo).
- Index Ride.status, Ride.createdTime, User.roles, and Payment.paymentStatus for fast queries.

State machine & enums

- Keep explicit enums: RideRequestStatus, RideStatus, PaymentMethod, PaymentStatus.
- Model ride lifecycle transitions explicitly and validate at service boundaries.

Event-driven design

- Use async events for notify/accept flows: RideRequestedEvent, DriverAcceptedEvent, RideStartedEvent, RideCompletedEvent, PaymentCompletedEvent.
- This enables horizontal scaling and resilient retries.

Saga pattern for payments

- Implement saga steps: reserve → assign → capture → settle. Compensate on failures (refund, cancel ride, notify user).

Observability & metrics

- Instrument: match latency, drivers notified per request, acceptance rate, payment failure rate, average time-to-assign. Add distributed tracing for request → match → accept.

Security & Privacy

- Hash passwords (bcrypt/argon2), store minimal PII in logs, mask PII, secure admin endpoints with RBAC and MFA.

5 — JPA / DB mapping hints (concise)

- Use @OneToOne for profile links with User. Mark user_id unique in child tables.
- Use @ElementCollection for roles or a separate join table depending on query needs.
- Use @Version for optimistic locking on RideRequest / Ride / Wallet.
- Use Point (geometry) types for locations with proper Hibernate spatial dialect if using PostGIS.

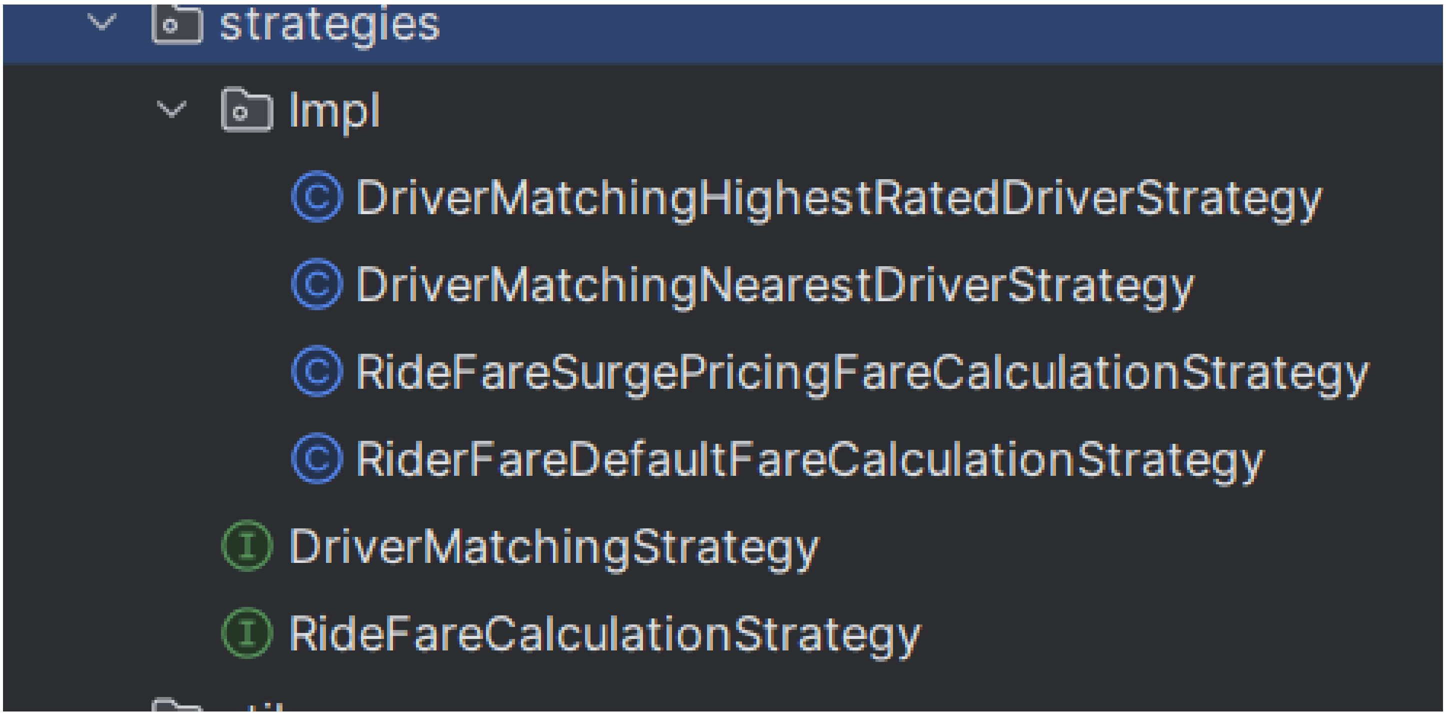
6 — Quick implementation checklist (next steps)

- Add enum definitions (RideStatus, PaymentStatus, etc.)
- Persist RideRequest early and implement transactional outbox.
- Implement strategy registry for fare & matching selection (avoid if/else).
- Add message bus for RideRequestedEvent and DriverAcceptedEvent.
- Implement atomic assignment/reservation (DB-level or separate service).
- Build payment saga skeleton and wallet transaction ledger.
- Add spatial index and scale test geo queries.
- Add metrics/tracing and unit + e2e tests for failure scenarios.

7 — Example state transitions (concise)

- RideRequest: PENDING → ASSIGNED → EXPIRED/CANCELLED
- Ride: ASSIGNED → ONGOING → COMPLETED → SETTLED
- Payment: PENDING → COMPLETED / FAILED (with compensation flow)

3. Adding Strategy classes



✅ High-Level Assessment of Your Strategy Design

Thecurrent implementation showcases a clean application of the **Strategy Design Pattern**, enabling dynamic, pluggable, and configurable business workflows for key Uber-like operations. This positions your backend to scale seamlessly as new market conditions or matching rules emerge.

1. Driver Matching Logic

- *DriverMatchingStrategy* → Core contract
- *DriverMatchingNearestDriverStrategy* → Matches based on proximity
- *DriverMatchingHighestRatedDriverStrategy* → Matches based on rating metrics

This design ensures your matching engine remains modular, testable, and easily extensible.

2. Fare Calculation Logic

- *RideFareCalculationStrategy* → Core contract
- *RiderFareDefaultFareCalculationStrategy* → Standard fare
- *RideFareSurgePricingFareCalculationStrategy* → Surge pricing logic

✅ Technical Excellence Highlights

🔹 Clear separation of concerns

Driver allocation logic and fare computation logic operate in independent strategy families. This reduces coupling and eliminates brittle code paths.

🔹 Pluggable strategy architecture

Adding new rules (e.g., AI-powered driver selection or ML surge prediction) becomes frictionless, requiring no changes to existing workflows.

🔹 Spring-friendly design

The annotated the matching strategies with @Service, positioning them for DI-driven runtime selection via:

- Qualifiers
- Strategy registry
- Factory pattern
- Config-based toggling

🔹 Clean DTO-driven contracts

Using RideRequestDto ensures The strategies operate on a business-facing request model instead of core entities, reinforcing domain boundary integrity.

🚀 Next Recommendations (Strategic Enhancements)

To unlock enterprise-grade scalability:

✅ Introduce a Strategy Factory

Centralize decision-making:

- Peak hours → Surge strategy
- Normal hours → Default strategy
- User-specific pricing → Loyalty strategy

✅ Integrate actual location & rating calculation

Instead of return List.of(), implement:

- Haversine distance for location
- Weighted rating models
- Filtering based on driver availability

✅ Enable configuration-driven strategy selection

Spring Profiles or database-driven configs to toggle strategies without redeployments.