Parking lot

Step 1
*3 flows <- Entry, Exit, Admin

Step 2
* Entry -> 1) Vehicle arrives
2) Based on vehicle Generate ticket
3) Allocate slot & floor
4) Mark that ↓ occupied
5) Return entry response

* Exit -> 1) Vehicle gives the ticket at counter
2) Calculate price <- hourly, flat
3) Process payment (generate receipt)
4) Release slot
5) Return exit response with receipt

* Admin -> 1) Can override (add/remove edit the slot)
2) Decides pricing amount for vehicles.
3) check parking slot periodically (manual/virtual)

Step 3 Identify core entities/model/Domain

1) TicketSystem — ID, vehicleId, SlotId, entry time, isActiveTicket
→ while exiting ↑
2) ReceiptSystem — ID, TicketID, exittime, payment status
1) vehicle system — ID, vehicle Num (license plate), type (bike, car)
2) parking System — ID, Slot (1-50), floor (1-5), isOccupied (yes after this vehicle)
5) pricing System — vehicletype, rateperhour, Days <- weekend high, weekday low
6) payment System — ticketID, Amount, gateway, status
↳ retrie logic
Adapter pattern <- razorpay, Stripe, UPI

* All SOLID principles applied, oops <- Interface, Encaps, abstraction

Step 4
Patter applied -> adapter -> Architecture Layer <- controller, service, repository

Step 5 Edge case -> missed ticked, Payment failure (retries)
* clock issues -> Do Centralized time service

Book My show (Movie/event ticket Booking)

Basic flow → user — city → movie → theatre → show → seat → payment → ticket(mail) notifying

Admin → admin access to add & remove → theatre, show, movie

Singleton pattern

UML Diagram

BookingService (Singleton)

- MovieController
- TheaterController
- paymentService

+ startBookingSession()
+ selectCity()
+ selectMovie(city)
+ selectShow(movie)
+ bookSeat(show)
+ generateTicket()

important method to run the application

Added only Attributes imaginarily add getters & setters for all attributes in all classes

Enum: SeatCategory { REGULAR, PREMIUM, VIP }

uses

uses

uses

Shows list of theatres when we select city

MovieController

- cityToMovies : Map
+ getMoviesByCity(city)
+ addMovies(movie, city)

TheaterController class

- cityToTheaters: Map
+ getTheaterByCity(city)
+ addTheater(theater, city)

uses BookingService

association (manages multiple movies but didn't own)

association (manages not owns)

PaymentService

+ ProcessPayment(amount)

Movie

- movieID
- movieName
- Duration

Theater

- TheaterID
- TheaterName
- Address
- city
- Screens : List
- Shows : List

Methods getters setters for all

Theater has multiple Shows, but theater can exist alone with shows

association

Composition (must have Screens)

Screen

- ScreenID
- Seats : List<Seat>

{ Enum of Seat }

Seat

int Seat
int row
Enum category

Composition

association

association

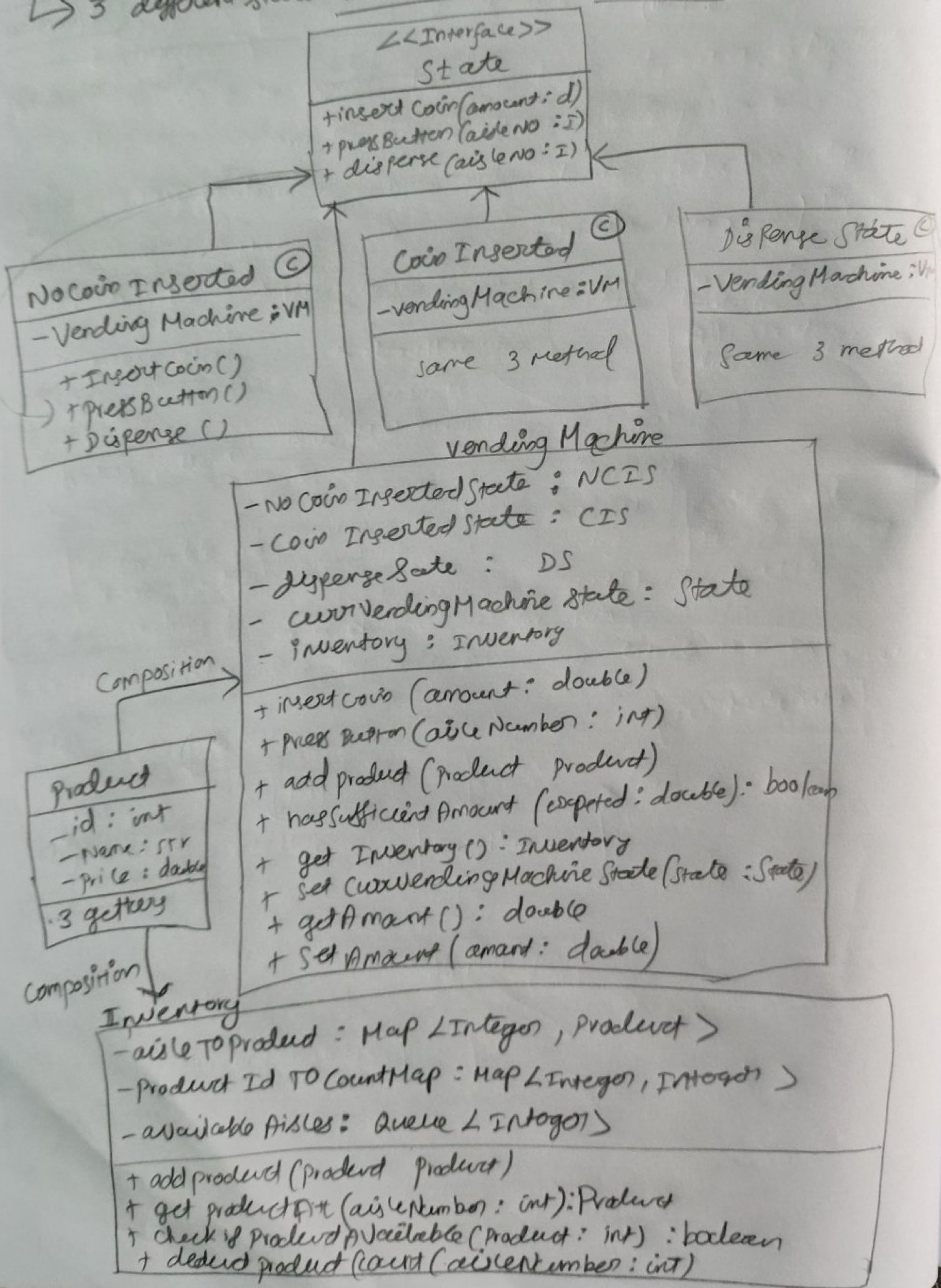
Show

- ShowId
- movie : Movie
- screen : Screen
- startTime
- bookedSeats : List

Association

(STATE PATTERN) Vending Machine

States → insertCoin() → pressButton() → Dispense() interface
↳ 3 different state i) NoCoinInserted, ii) inserted iii) Dispense



Strategy, Singleton
Factory like
SOLID, Enums

Ride Booking System

Rider → Entity to persist rider's name, id, rating

Rider Manager → Store the info of all the riders (map of rider ID → Rider)
↳ Singleton

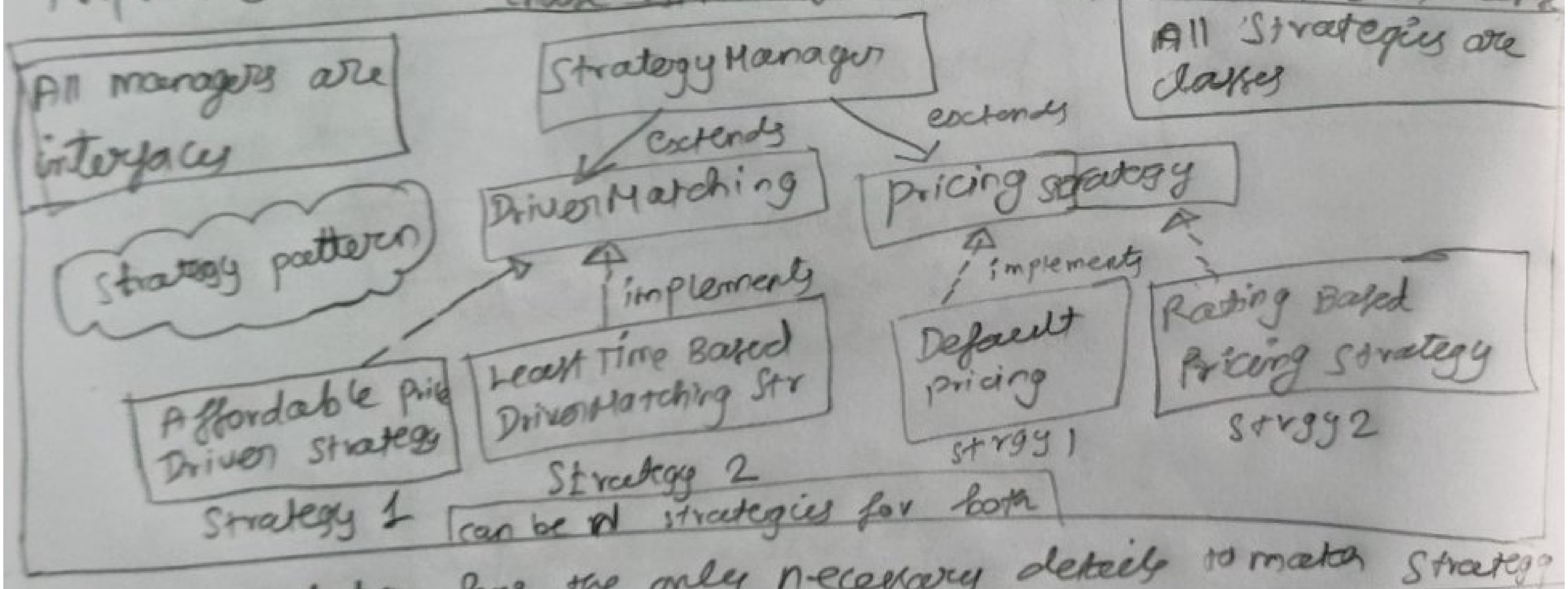
Driver → Entity to persist driver's name, id, rating, isAvailable(ready to take trip)

Driver Manager → Store the info of all the Driver (maps with driverID → Driver)

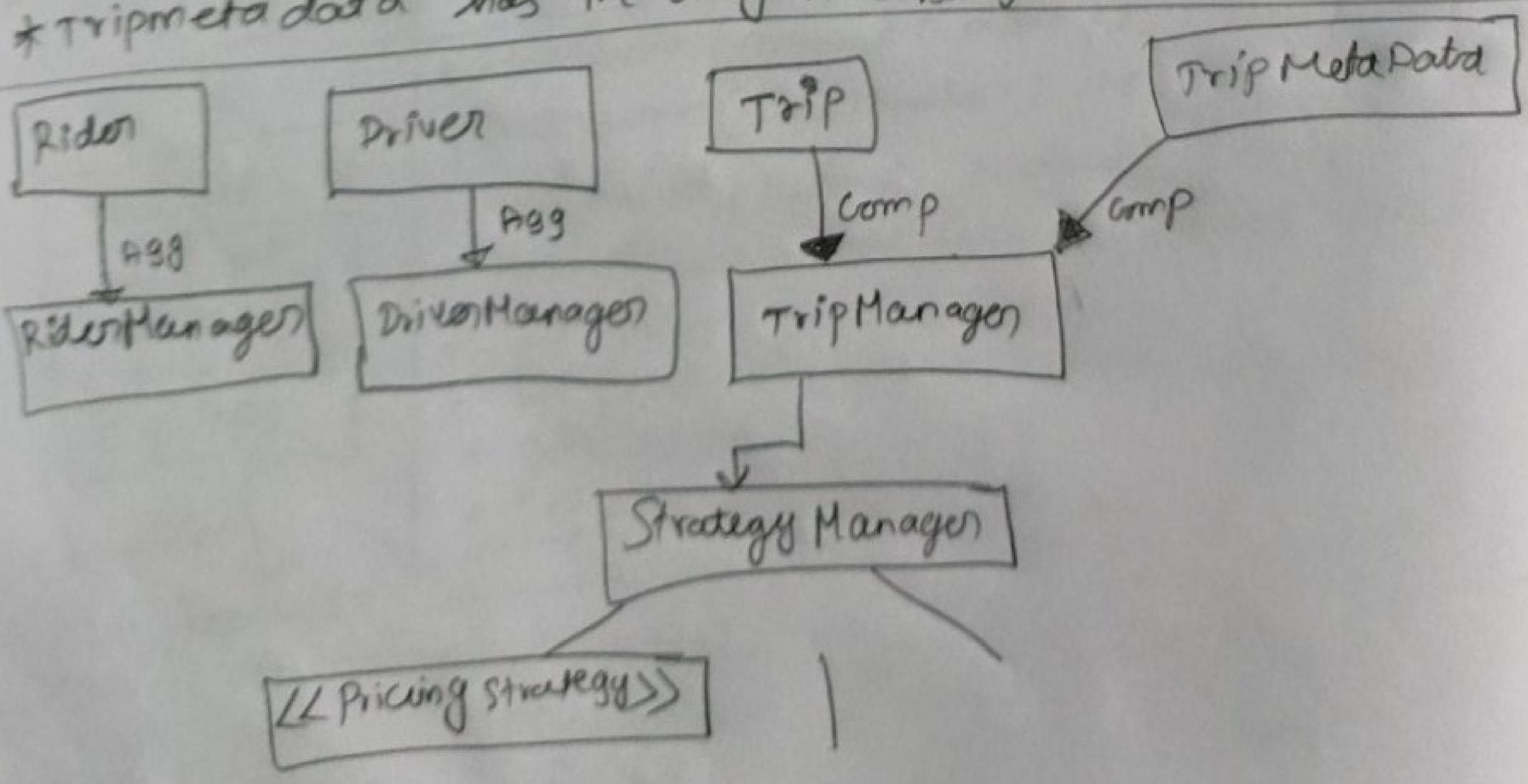
Trip → Trip Creation we have rider, driver, srcLocation & destination
(Respons to Create trip)

TripManager → Manages all trips with tripMetadata + Trip

TripMetaData → Has only the essential info needed to create trip to choose strategy (rider, driver (not DOB, Licence etc.))



*Tripmetadata has the only necessary details to make Strategy



Notification System

Design structure i) plug & play model, ii) Extendable - SMS, EMail, Popup
iii) Notification grows dynamically, iv) Store all notification/logging

* used Decorator, Strategy & observable pattern.