If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**
https://www.linkedin.com/in/yadav-sanjay/ https://www.youtube.com/@GeekySanjay

Backend LLD-3: Machine Coding - 2: Design TicTacToe - Feb 06

- **Cordinaty revision**
- **How to approach schema design**
- **How to start code**
- **MVC Architecture**
- **How to code**
- **Design tic-tac-toe**
    - **UNDO**

Cordinaty revision -> When dealing with relationships between tables in a database schema, especially in the context of 1-to-1 and 1-to-many relationships, coordination becomes crucial for maintaining data integrity. Let's explore these relationships

One-to-One Relationship:
In a one-to-one relationship, each record in the first table is associated with at most one record in the second table, and vice versa.
Coordination involves ensuring that the relationship is properly established, often through the use of a shared key or a foreign key constraint.
Example: Consider a schema with tables "Employee" and "EmployeeDetails" where each employee has a unique employee ID, and the EmployeeDetails table holds additional information for each employee.

One-to-Many Relationship:
In a one-to-many relationship, each record in the first table can be associated with multiple records in the second table, but each record in the second table is associated with at most one record in the first table.
Coordination involves establishing a foreign key in the "many" side of the relationship, pointing back to the primary key in the "one" side.
Example: Consider a schema with tables "Department" and "Employee" where each department can have multiple employees, but each employee belongs to only one department.

Many-to-Many Relationship:
In a many-to-many relationship between tables in a database schema, each record in one table can be associated with multiple records in another table, and vice versa. To represent such relationships, a junction or associative table is often introduced. This table serves to connect the two entities, establishing the many-to-many relationship. Let's explore this concept with an example:
Consider a scenario where you have two entities: "Students" and "Courses." A student can enroll in multiple courses, and a course can have multiple students. Here's how you might design the schema to handle this many-to-many relationship:

Cardinalities revision

Four types

1:1

1:m

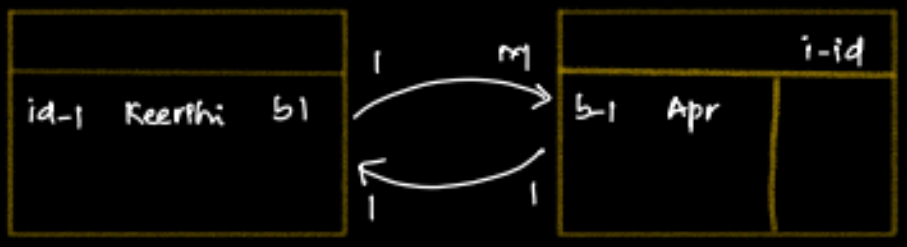
m:1

m:m

1:1 : How to represent?

Put id of one table in another table.

1:m or m:1

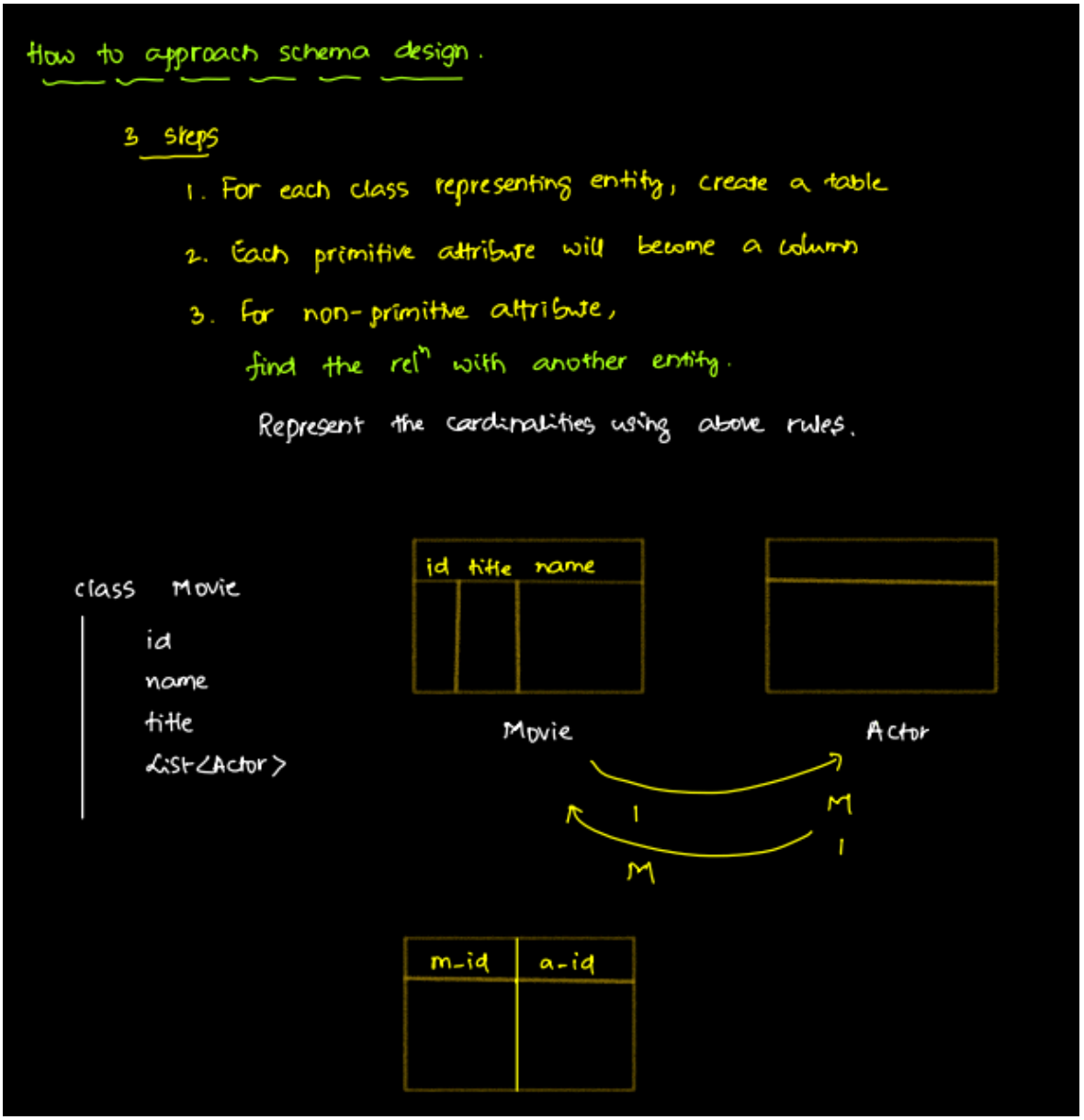


Put id of '1' side in 'm's side.

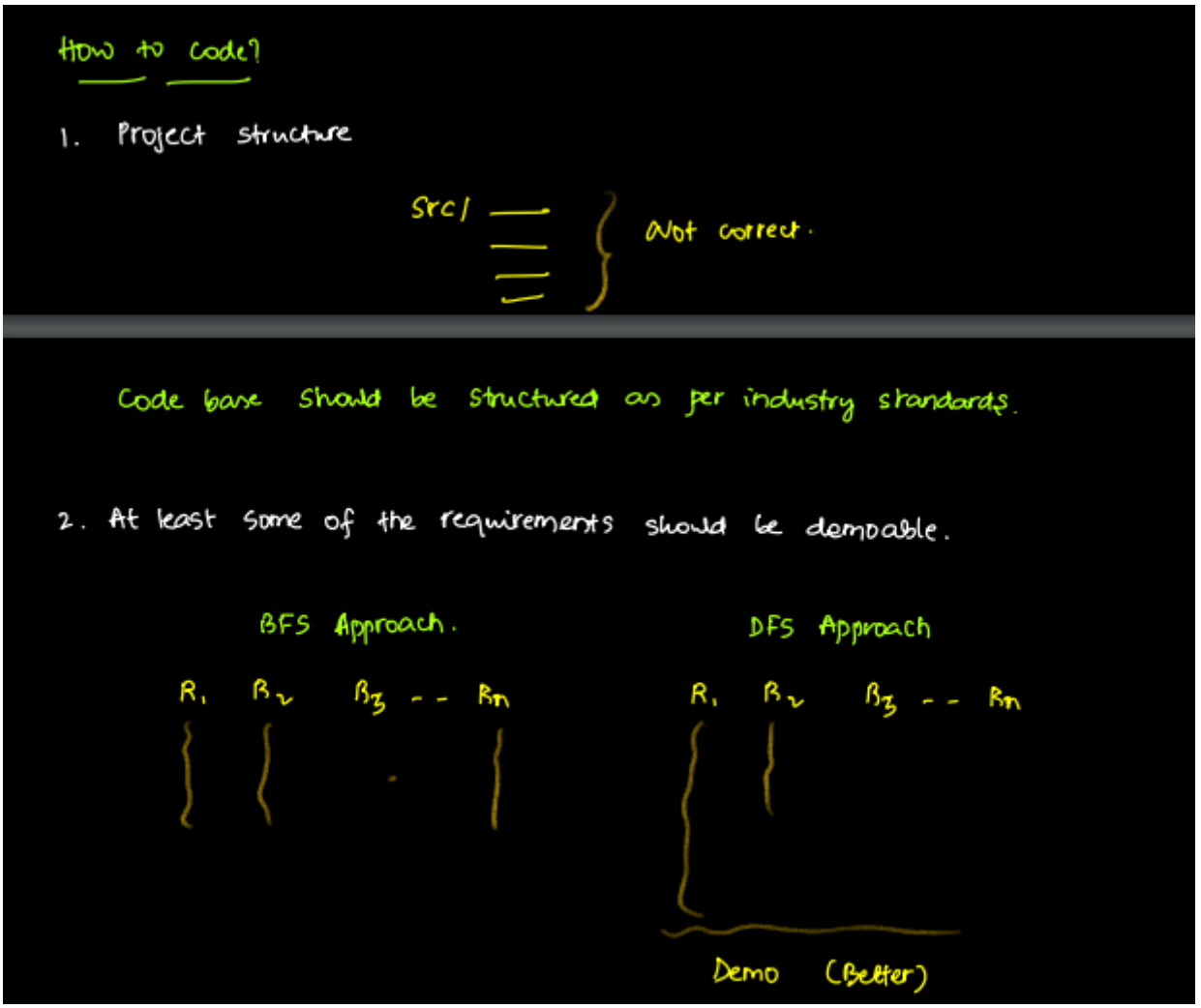
m:m

-> Create a separate mapping table.

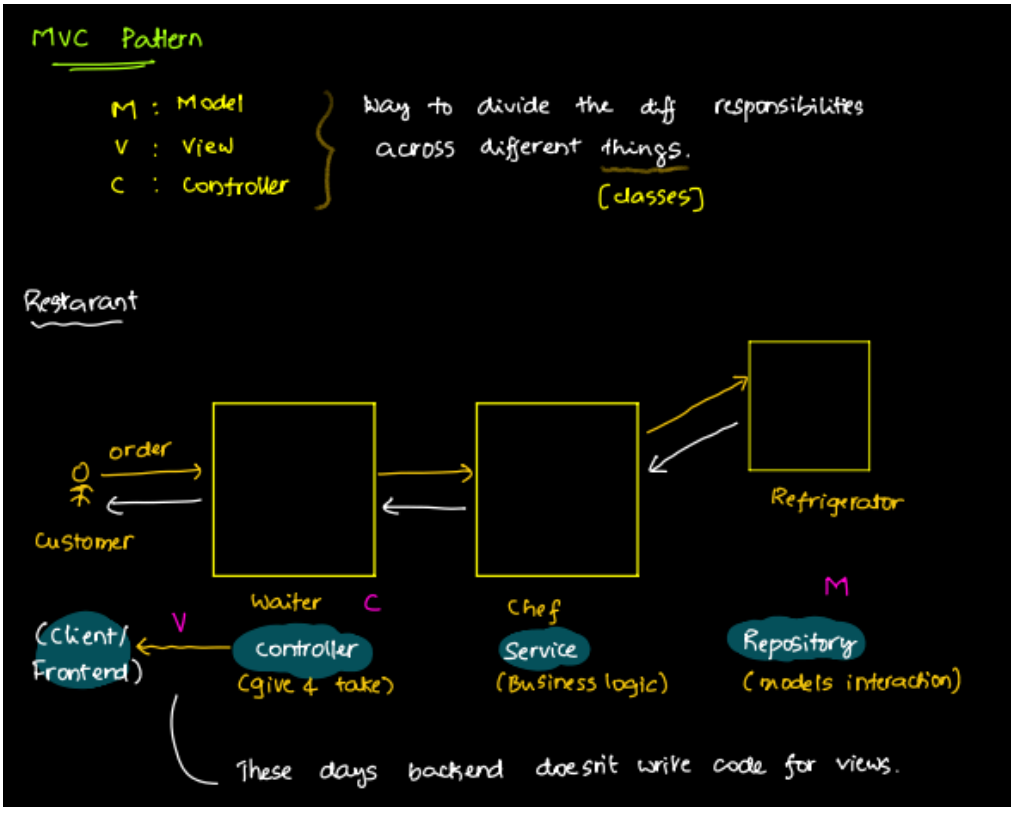
How to approach schema design -> Designing a schema means organizing information based on a class or category. For example, if we have a class called "movie," which represents entities (**any living or non-living thing with specific features and actions its called)**, we create a table for it. The table includes premetive data memebers like "id," "name," and "title" as columns. For more complex data, such as non premetive data lists (like a list of actors), we determine how it connects with the main table. If, for instance, movies can have many actors, and actors can be in many movies (Many-to-Many or M:M), we include this relationship in our schema. This way, we build the structure of our database based on the characteristics outlined in the given class.



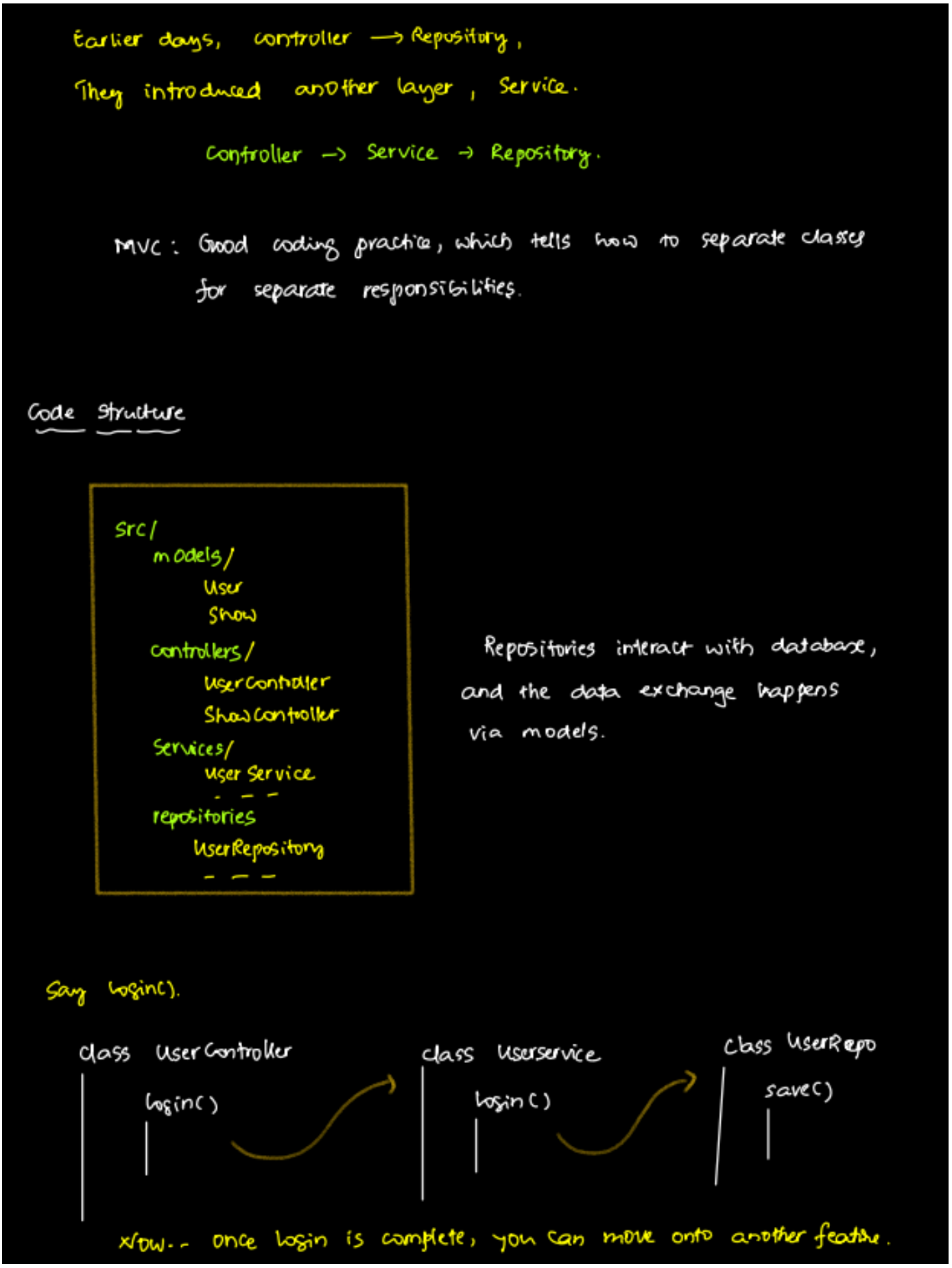
How to code -> When writing code, it's helpful to use a standard project structure like MVC (Model-View-Controller). Also, it's a good idea to follow a DFS approach, which means not working on all features at once. Instead, focus on finishing one feature completely, make it demonstrable, and then move on to the next feature. This way, you can build and test your code step by step, making the development process more organized and manageable.



MVC Architecture -> MVC (Model-View-Controller) is a design pattern commonly used in software development, including Java applications. It separates an application into three interconnected components: Model, View, and Controller.



When integrating services into the MVC architecture, the pattern often evolves into something referred to as the "Service-oriented architecture (SOA)" or "Model-View-Controller-Service (**MVCS**)." Services encapsulate additional business logic and functionality, providing a way to modularize and decouple the application further



Here's an explanation:

Model:
The Model continues to represent the application's data and core business logic.
In addition to handling data manipulation, it may also interact with services to perform specific operations or retrieve additional information.
The Model remains responsible for maintaining the state of the application.

View:
The View is responsible for presenting the data to the user and receiving user input, just as in the traditional MVC pattern.
Views can also interact with services for additional data or operations, especially in scenarios where complex business logic is involved in rendering the user interface.

Controller:
The Controller acts as an intermediary between the Model, View, and Services.
It receives user input, updates the Model, and interacts with services to perform business operations.
The Controller orchestrates the flow of data between the Model, View, and Services, ensuring that the application's logic is properly executed.

Service:
Services encapsulate specific business logic and functionality that may not be directly tied to a particular Model or View.
They can handle tasks such as authentication, data validation, communication with external APIs, or any other application-specific functionality.
Services promote code reusability and maintainability by centralizing specific functions that multiple parts of the application may need.

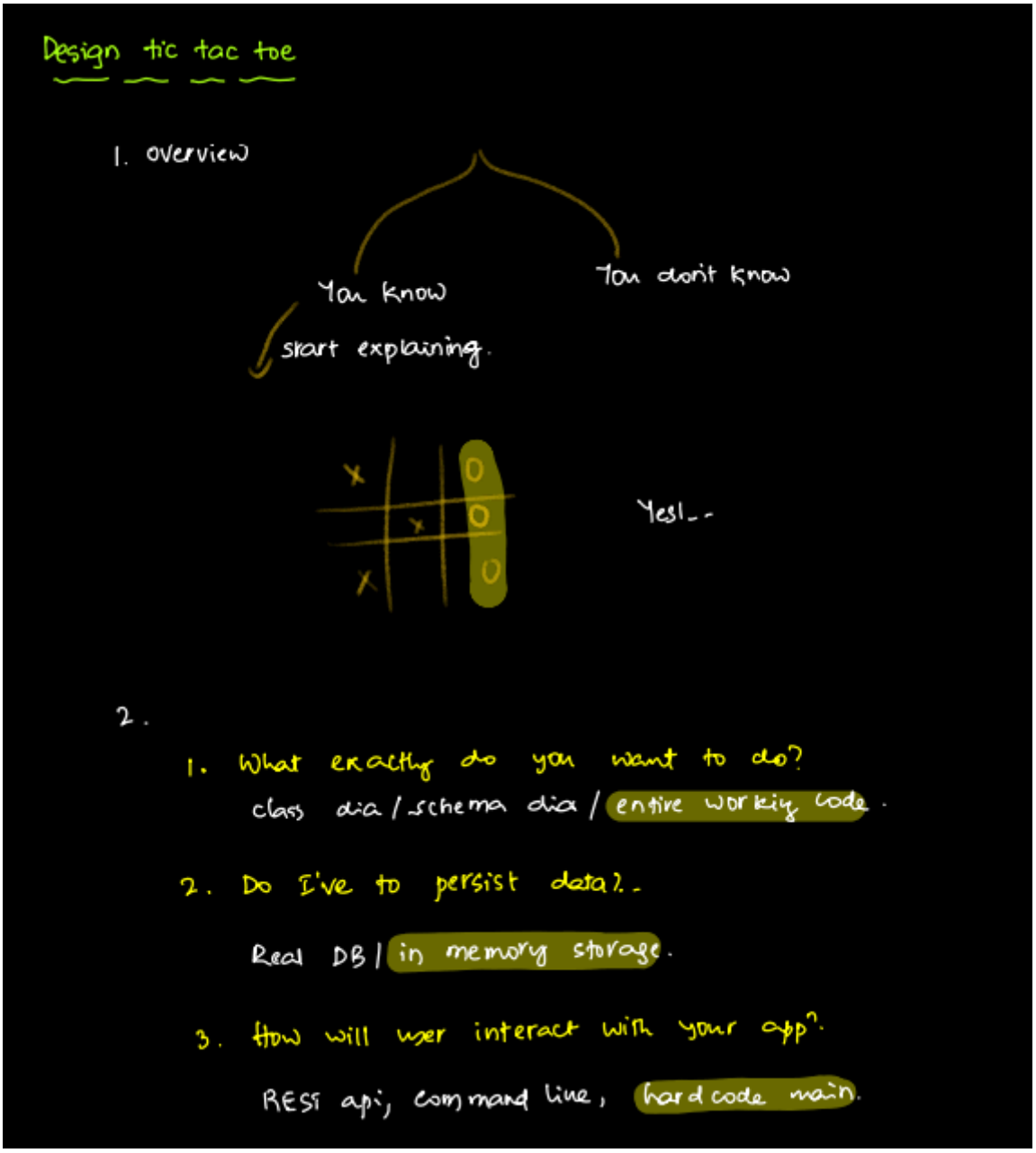
```
// Model
public class Student {
    private String name;
    private int age;

    // Getters and setters...
}

// View
public class StudentView {
    public void displayStudentDetails(String name, int age) {
        // Display student details in the GUI
    }
}
```

```
// Service
public class StudentService {
    public void performAdditionalLogic(Student student) {
        // Additional business logic specific to Student
    }
}

// Controller
public class StudentController {
    private Student model;
    private StudentView view;
    private StudentService service;

    public StudentController(Student model, StudentView view, StudentService service) {
        this.model = model;
        this.view = view;
        this.service = service;
    }

    public void setStudentDetails(String name, int age) {
        model.setName(name);
        model.setAge(age);

        // Perform additional logic through the service
        service.performAdditionalLogic(model);
    }

    public void updateView() {
        view.displayStudentDetails(model.getName(), model.getAge());
    }
}
```

In this example, the StudentService class represents a service that encapsulates additional logic related to the Student entity. The StudentController can utilize this service to perform tasks beyond the basic Model-View-Controller interactions. The incorporation of services enhances modularity and allows for better organization of code.

What is API -> API stands for Application Programming Interface. Think of it like a contract: you provide a specific input, and in return, the API gives you a particular output. In simpler terms, an API is like a bridge between two applications. You send a request to it, and it sends back a response, functioning as a communication interface between different software components.

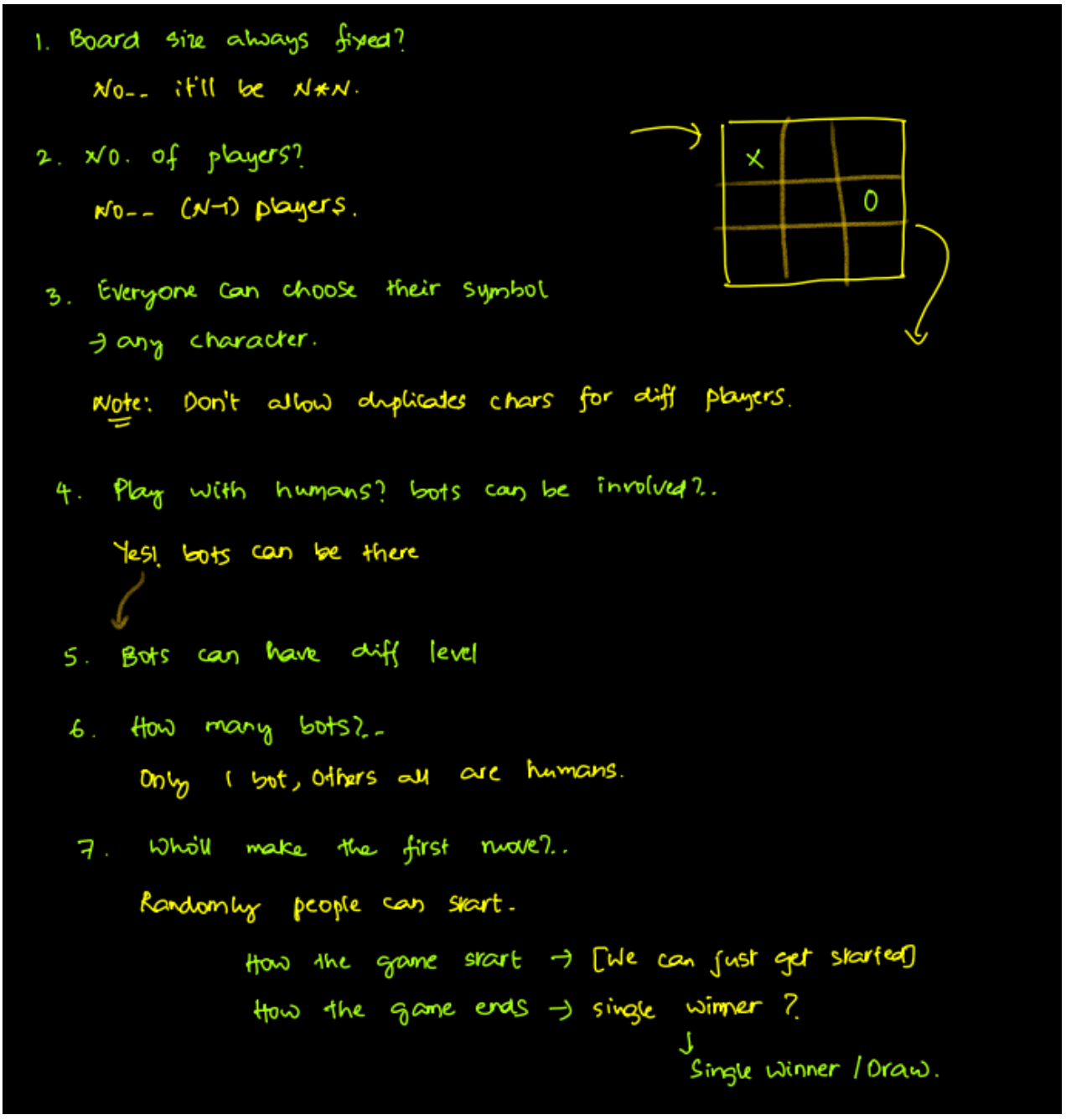
Design tic-tac-toe



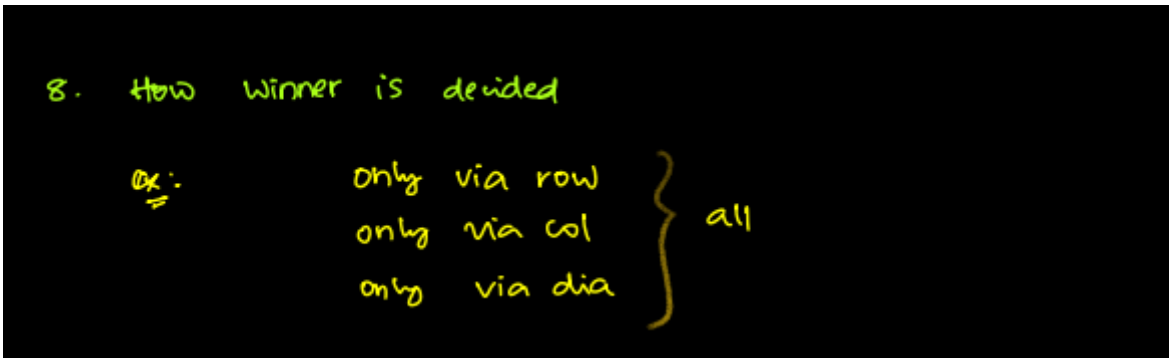
Before initiating any project, it's essential to follow specific steps, as depicted in the above illustration. **First**, provide an overview of the project. In this initial phase, there are two scenarios: you might already have information about the product, or you might not. If you have the details, confirm your understanding with the client. If not, politely inquire for clarification.

The second step involves asking the interviewer or client about their specific requirements. Offer suggestions and inquire about expectations—whether they're looking for a class diagram, a schema diagram, or the entire workflow. If it involves creating class and schema diagrams, start gathering requirements. If it's about the entire flow, ask additional questions before gathering requirements. “**Do i have to persist data**” like DB or in Memory storage. next qution **“How will user interact with your app”** like REST api, Command line or hardcoded value.

Requirment gathering ->
Initiate the requirement gathering by proposing specific questions. visualize the game and start from outside to inside. Certain common queries can be employed to extract essential requirements for games such as Chess, Snake & Ladder, and Ludo, as illustrated below like what is the board size it will be 3X3 or any custom dimention.

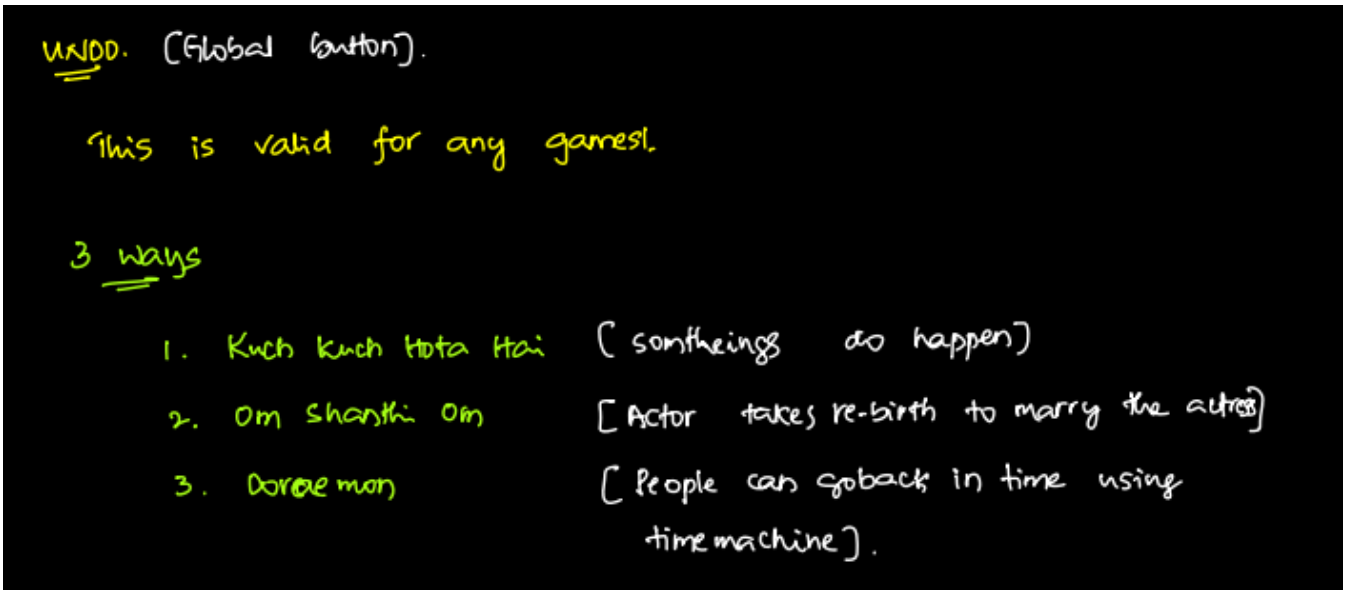


Next, we can inquire about how the winner is chosen. When developing a game, aim for flexibility. Demonstrating this capability involves asking about the different ways a player can win.

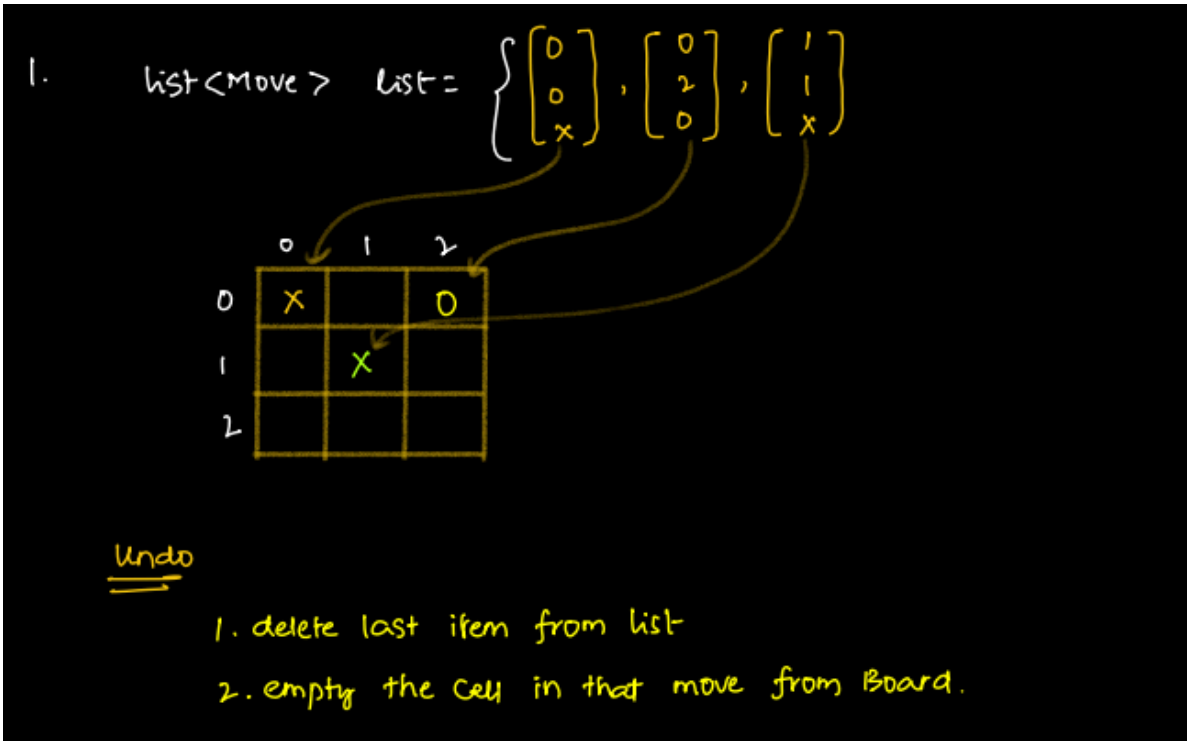


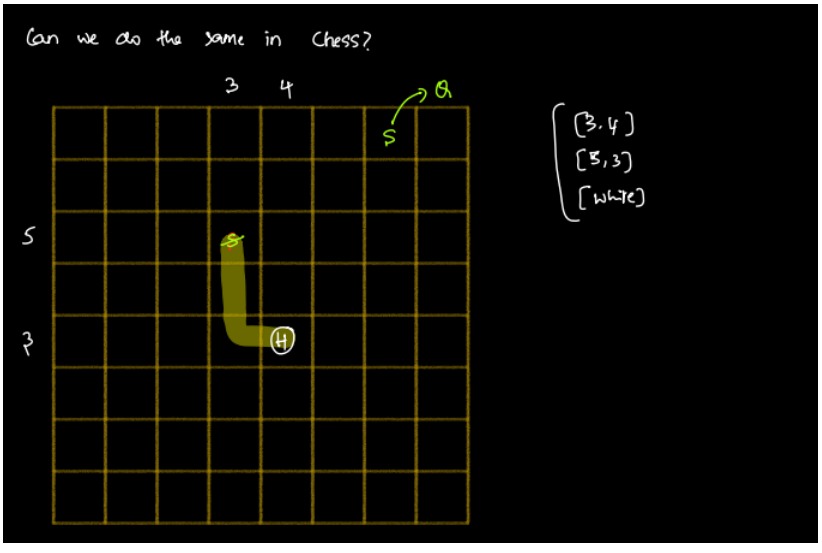
Common requirment to any game - below question can we ask for any game because this are the common for all.

Undo -> It is the global undo means it will affect direct main game state. to understand we have tree ways which we can use **kuch kuch hota hain**, **Om santi om** and **Doraemon.**

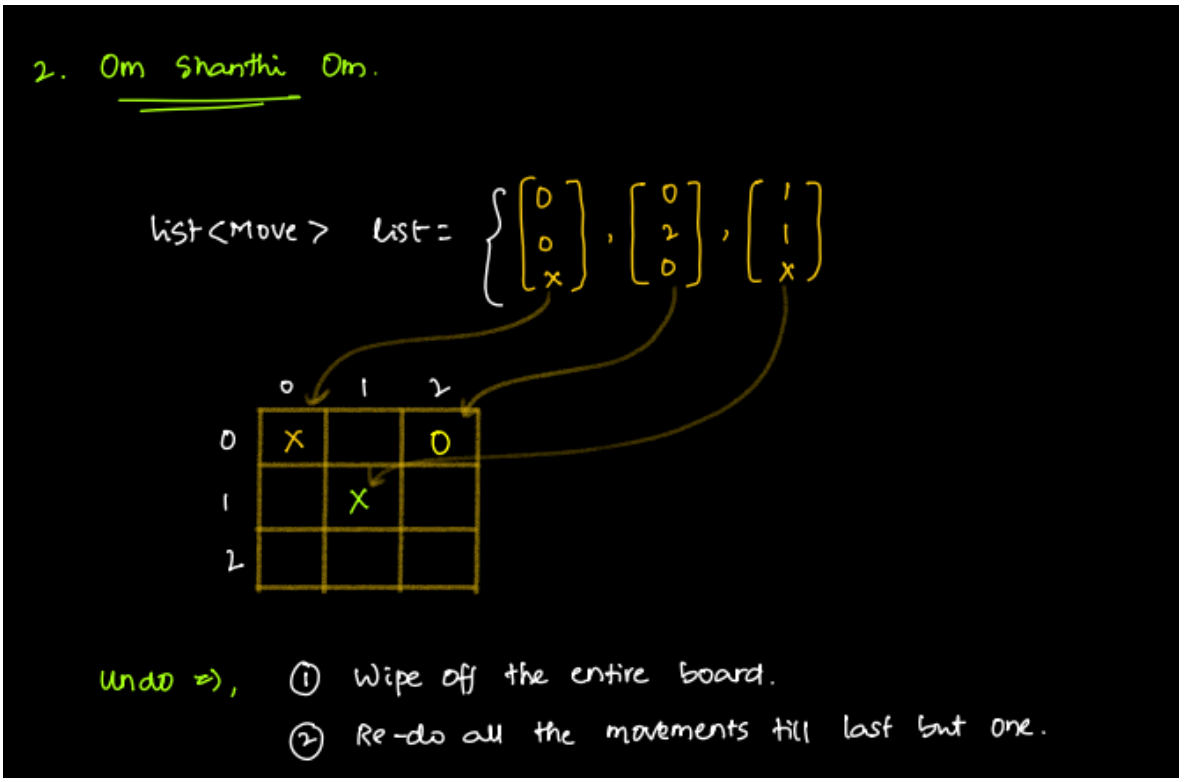


Kuch Kuch Hota Hai -> It's like removing the last step, but it doesn't work for all games, such as chess, where recreating the previous state is not possible.

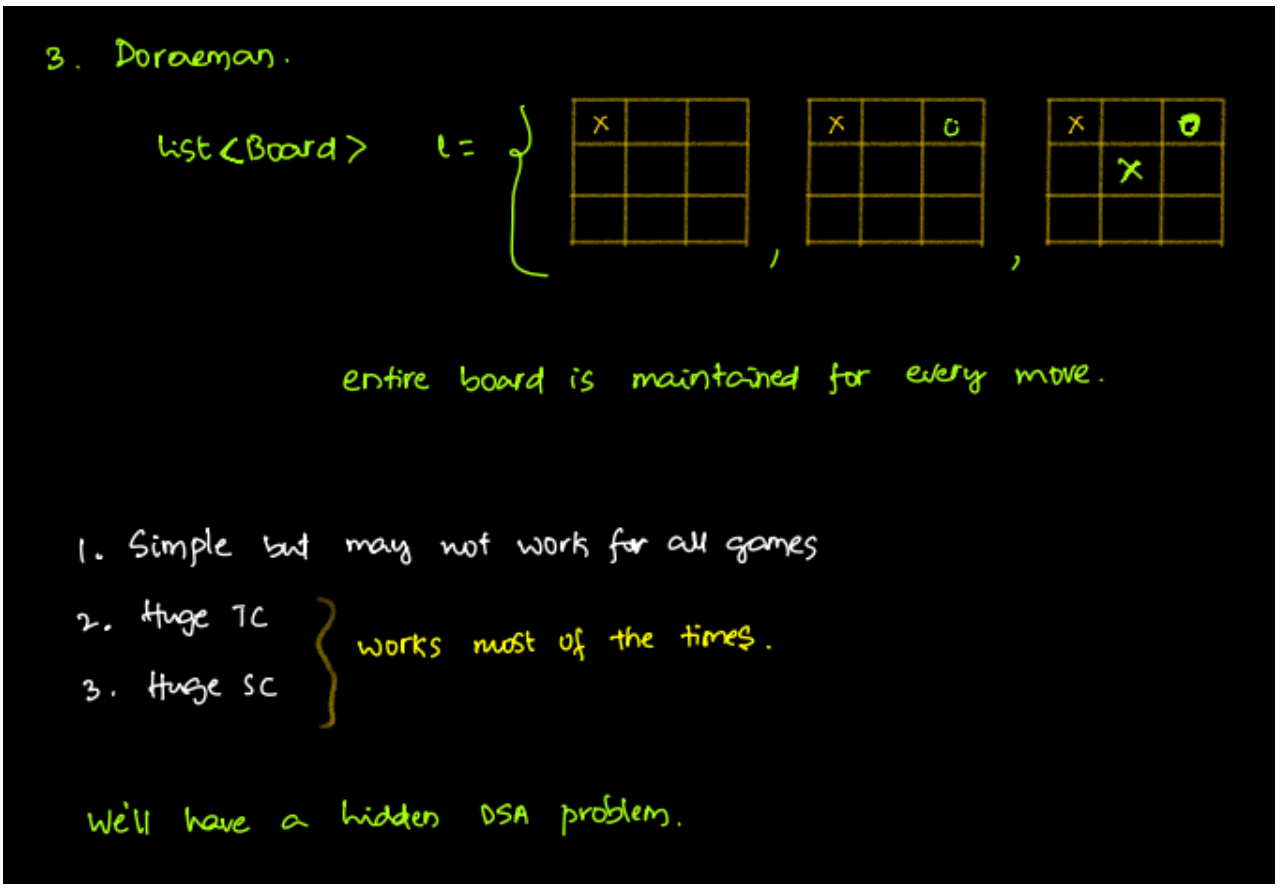




Om Shanti Om (Actor takes re-birth to marry actors) -> We recreate the entire board except for the last move. This method works for all games but can be quite expensive and huse Time complexity.



Doraemon (People can go back in current time using a time machine) -> Here, Instead storing moves. we save the complete board in List after every move, making it possible to retrieve it with O(1) complexity, though it comes at the cost of more space.



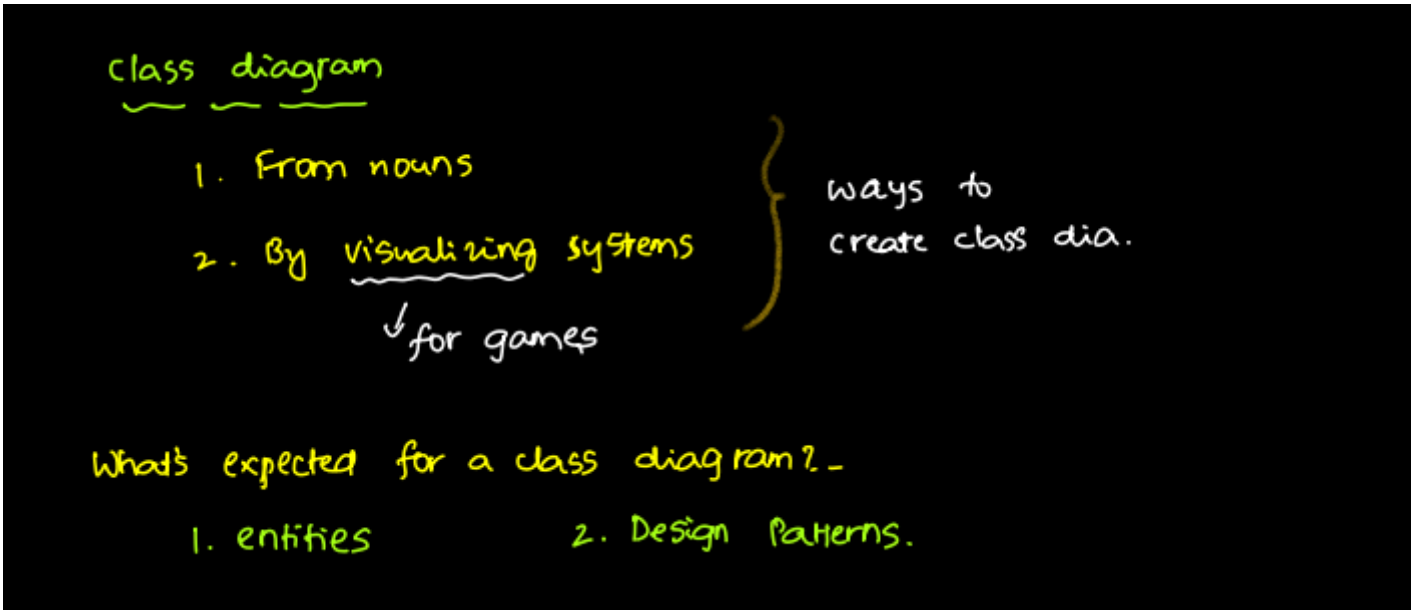
Backend LLD-3: Machine Coding - 3: Design TicTacToe - Feb 08

- **Class Diagram**
- **Summarising the design patterns that we will be using**
- **How to find the winner**
- **Finding Winner O(1)**

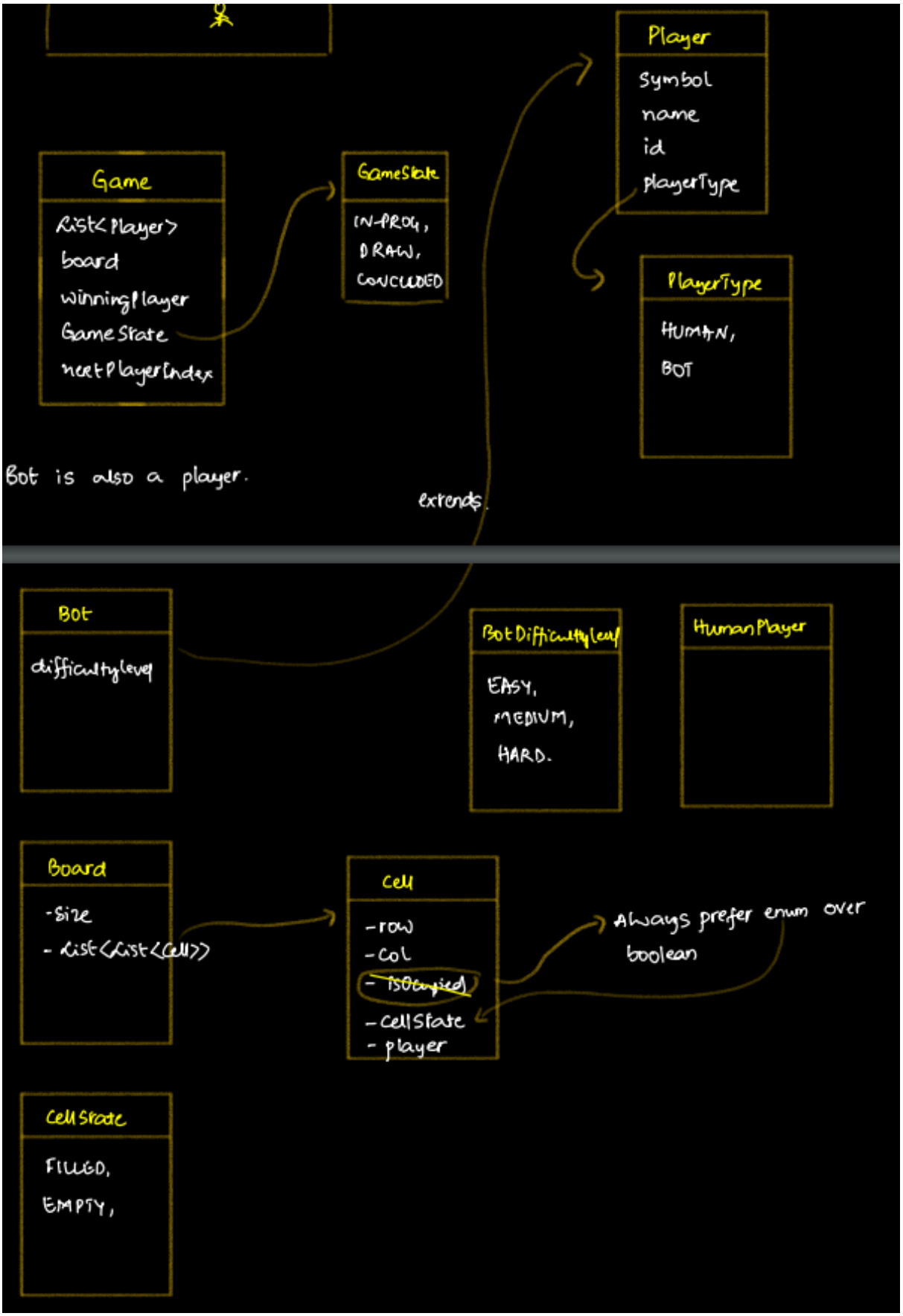
Class Diagram ->

Creating a class diagram can be approached in two ways: either by identifying nouns or by visualizing the system. When requirements are specified, it is advisable to begin with the noun-based approach. However, in the absence of specific requirements, starting with the visualization approach is also effective, particularly for games where a visual representation is crucial.

A comprehensive class diagram should encompass two essential elements: entities and design patterns.



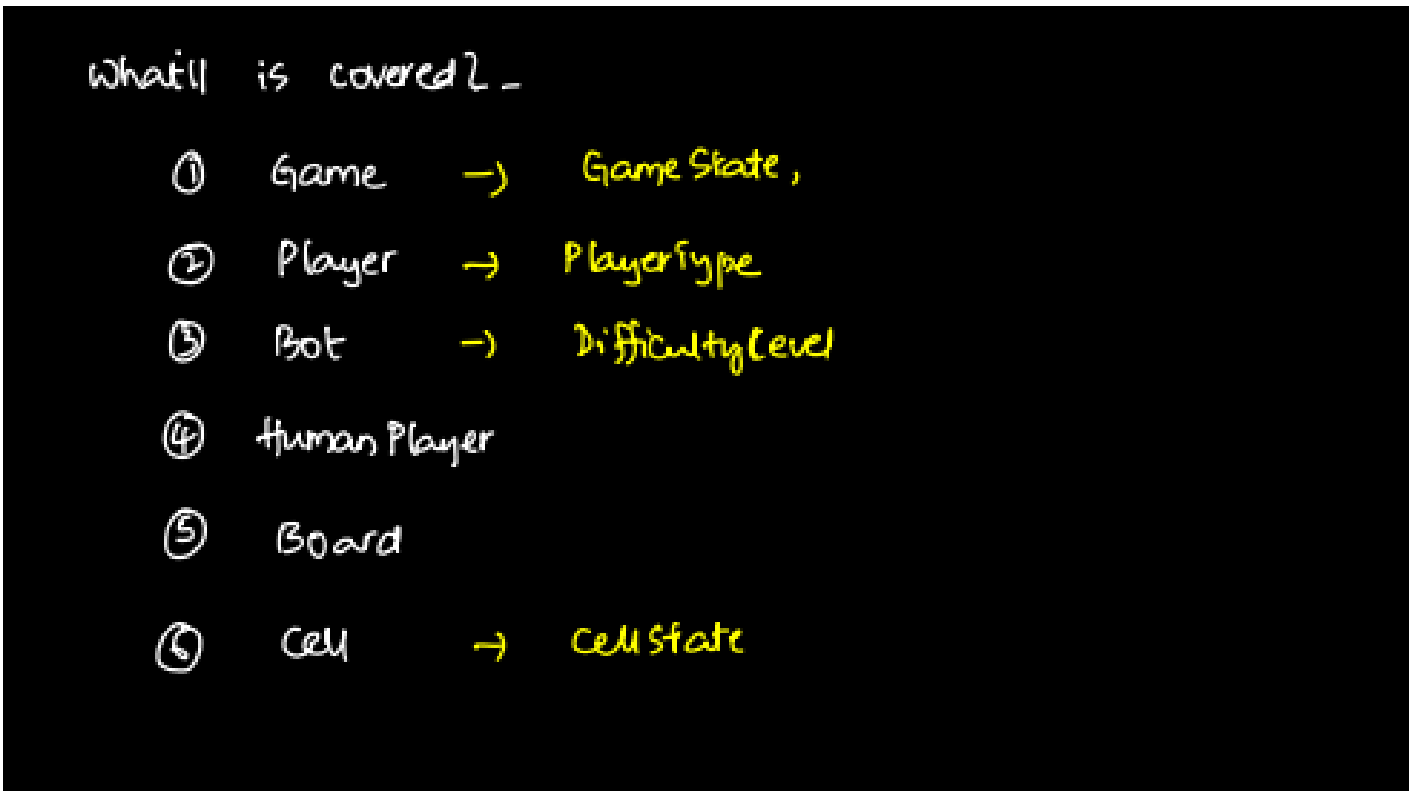
During the initial stages of developing a class diagram, it is essential to consider **whether any storage** is required **while moving from the outside to the inside**. As the thought process progresses inward, three fundamental entities typically emerge: player, board, and game. Establishing classes for each entity allows for a more systematic exploration of their internal components. For instance, within the **game class**, one might identify a list of players, the **board**, the **winningPlayer**, the **GameState**, and the **nextPlayerIndex**. This systematic breakdown facilitates a thorough understanding of the internal structure of the system.



Subsequently, we will establish an ENUM named GameState with three possible states: **IN_PROG** for games in progress, **DRAW** for drawn games, and **CONCLUDED** for completed games. Additionally, we will introduce a **Player class**, featuring properties such as **symbol**, **name**, **id**, and a crucial attribute named **playerType**. Given that there are two distinct player types – human and bot – we'll employ an ENUM to represent these types, defining **HUMAN** and **BOT** as the possible values for the playerType property.

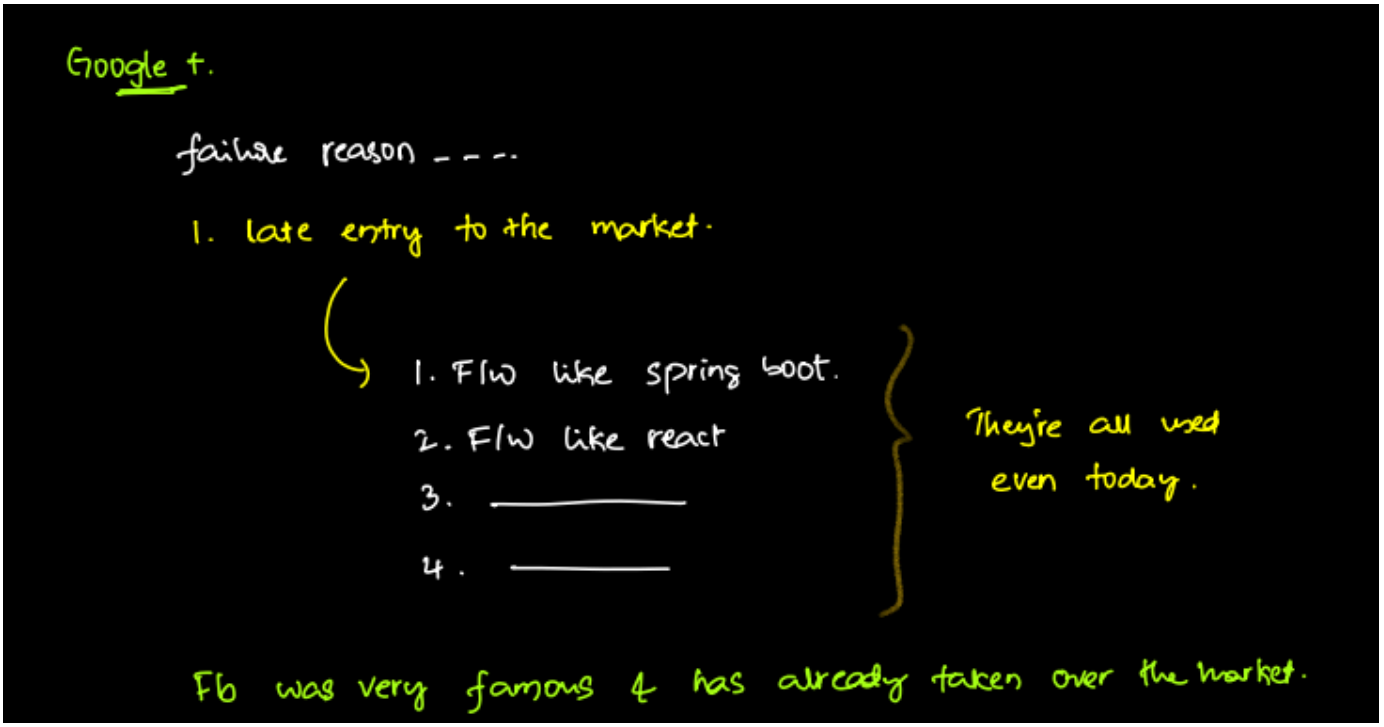
The Bot is considered a player, and it will inherit from the Player class. Additionally, it will introduce a new property called DifficultyLabel, which is an enumeration denoted as an ENUM with options including **EASY**, **MEDIUM**, and **HARD**.

Now that we've finished defining the Player class, we're ready to move on. The next step is to create the **Board** class, which will have properties like **'size'** and a **list of lists representing cells**. Each cell will have characteristics such as **'row'** and **'col'** for identification, and its state will be determined by whether it is occupied or not. The question arises: should 'isOccupied' be a yes/no (boolean) or something else? For better coding practices and to accommodate future needs, we'll use an ENUM named **CellState** with options like **'Filled'** and **'Empty'**. This approach is preferred over using a simple yes/no (boolean) as it allows for better scalability and the possibility to add more states later on. at the end we will add Plyer object.



We'll review the list of requirements to ensure that everything is covered, and then we'll document all the entities along with their interfaces.

Story of Google+ -> The lesson from this narrative is that no system is flawless, and attempting to achieve perfection is not advisable. This is evident in the case of Google Plus, which failed because it entered the market late because they launch many tools which used in google plus like spring boot etc. According to studies, the pursuit of perfection led to a delayed entry, allowing other tools like Facebook to establish themselves in the market successfully. If Google+ had launched a few years earlier, it might have experienced success instead of failure.



UNDO -> After reviewing all the requirements, there is one outstanding task: UNDO functionality. To implement the UNDO feature, we need to track all moves. To achieve this, we will introduce a new class called "Move," which includes attributes such as Cell, Symbol, and Player. This Move class will be integrated into the Game class. To enable UNDO, we will adopt the "Kuch Kuch Hota Hain" technique, where we simply remove the last step in the Move class. This process involves creating a mechanism for reverting the game state by deleting the most recent move.

==========

Tic Tec Toe in Action!

Download repository from the video description

https://www.youtube.com/watch?v=B5zZa9vN4yo

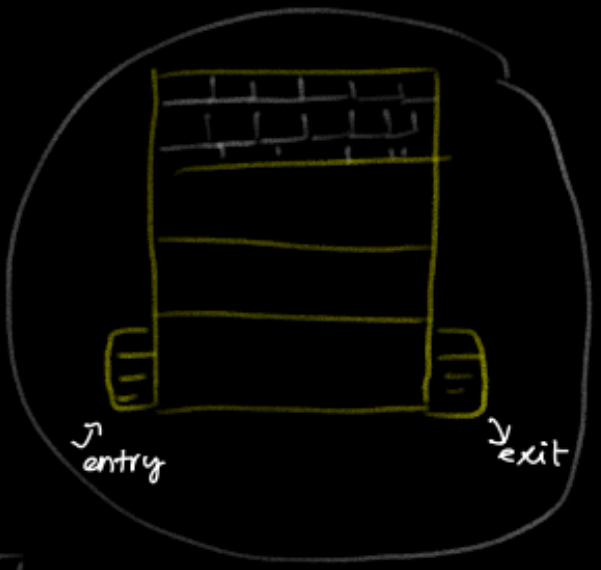
Like | Comment | Subscribe

==========

Visualise

ParkingLot

- list<floor>
- list<entry/exit> gates
- allowedVehicleType
- status



ParkingFloor

- list<parkingSlots>
- floorNumber

Gate

- type
- number
- operator
- status

ParkingSlot

- vehicle type
- parking status
- number (slot)
- floor

Operator

- empId
- name

Vehicle

- vehicle type
- number
- nameOfOwner
- ownerContact

Ticket

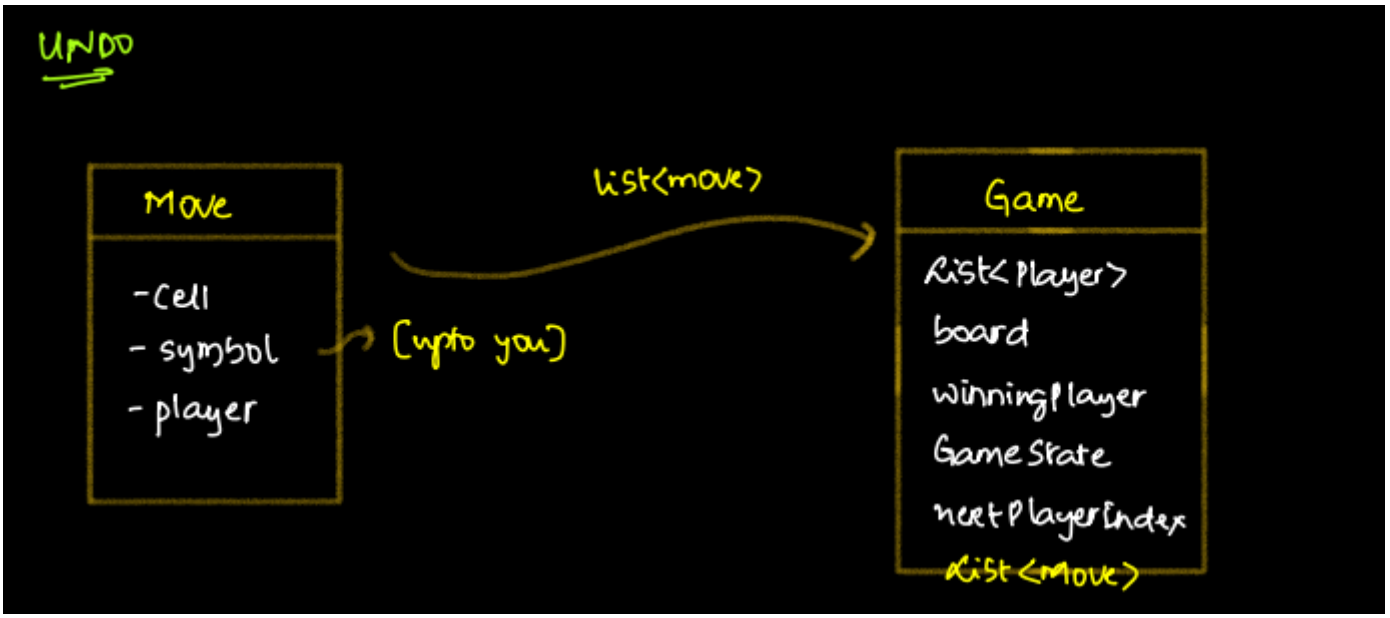
- number
- entry time
- vehicle
- parkingSlot
- gate
- operator

Bill

- exitTime
- ticket
- amount
- gate (exit)
- operator
- status (paid/not)
- payment

Payment

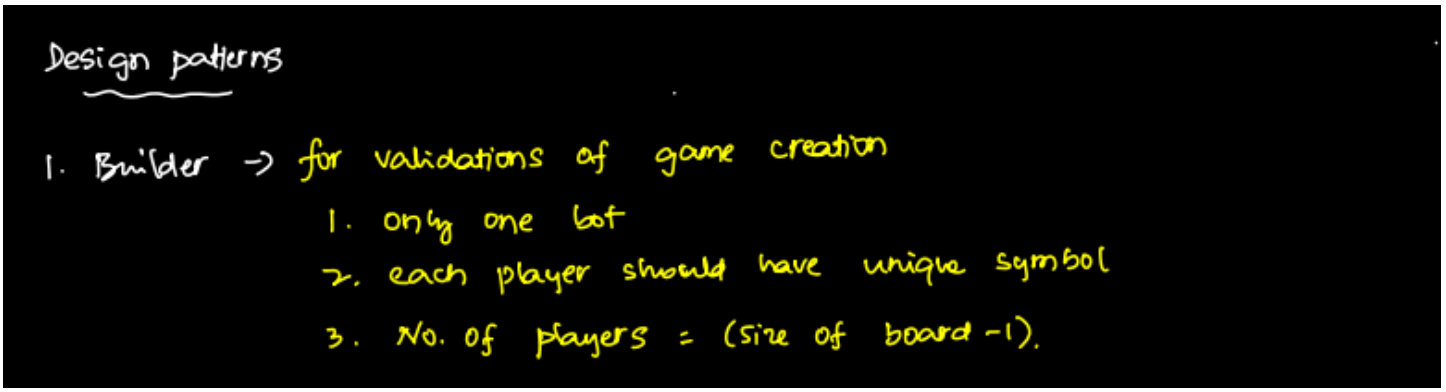
- refNumber
- status
- time
- mode

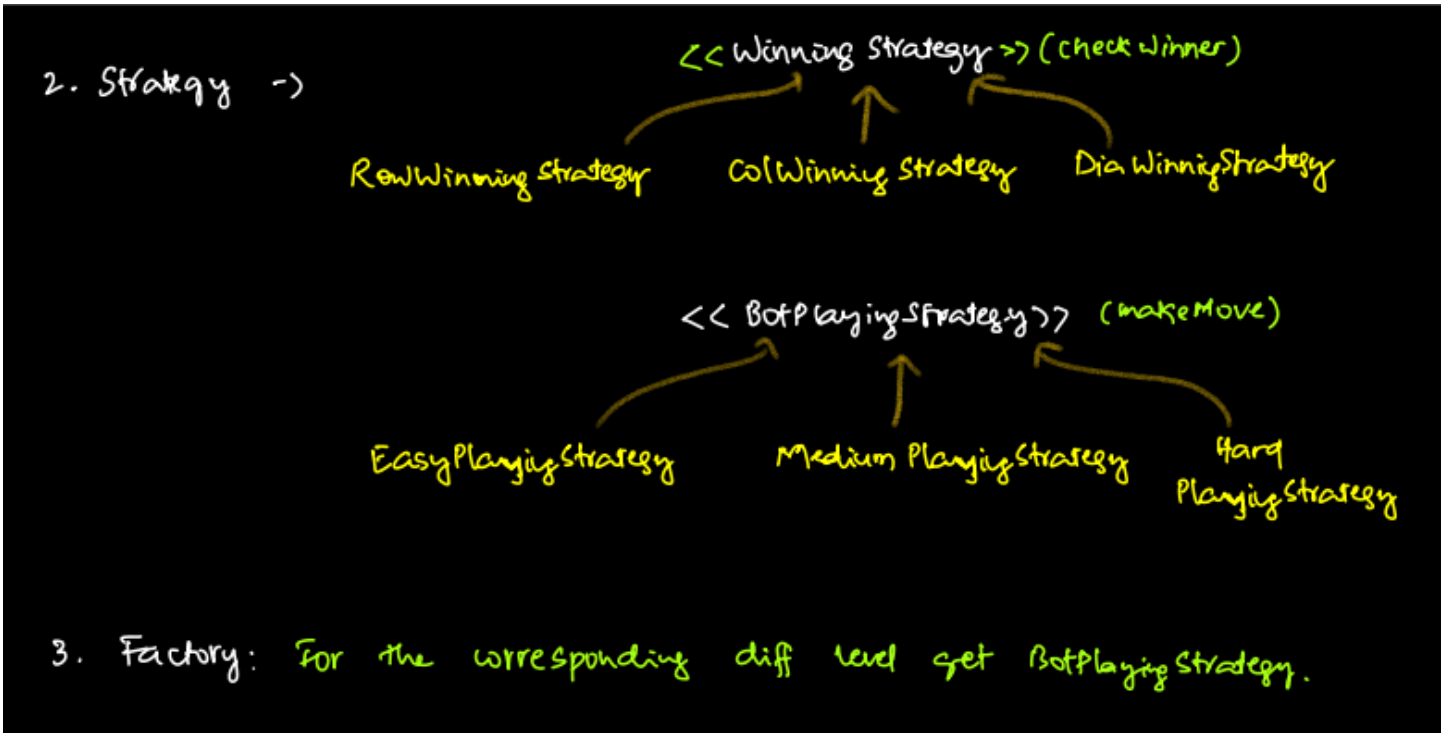


Summarising the design patterns that we will be using ->

To identify design pattern requirements, we will carefully review each requirement individually. During this analysis, we may recognize certain patterns, such as encountering a validation issue during object creation, such as '**preventing duplicate characters for different players.**' This observation indicates a need for the **Builder pattern**, as it effectively addresses our validation requirements during the object creation process and **To address the requirement of returning distinct Strategy depending on the lavel**, we will employ the **Factory Pattern**.

Subsequently, we come across another requirement: **"Bots can have varying difficulty levels."** and "**How winner is decide based on row,coulmn"** According to this specification, the code might undergo significant changes based on the bot's level. This situation aligns with a pattern-like problem, where different tasks need to be performed based on changing circumstances. In such cases, we will employ the **Strategy Pattern** to address the dynamic nature of the difficulty levels.





==========

Unlock IntelliJ Ultimate for Free with Exclusive Scaler Discount Code - Download Now!

Visit

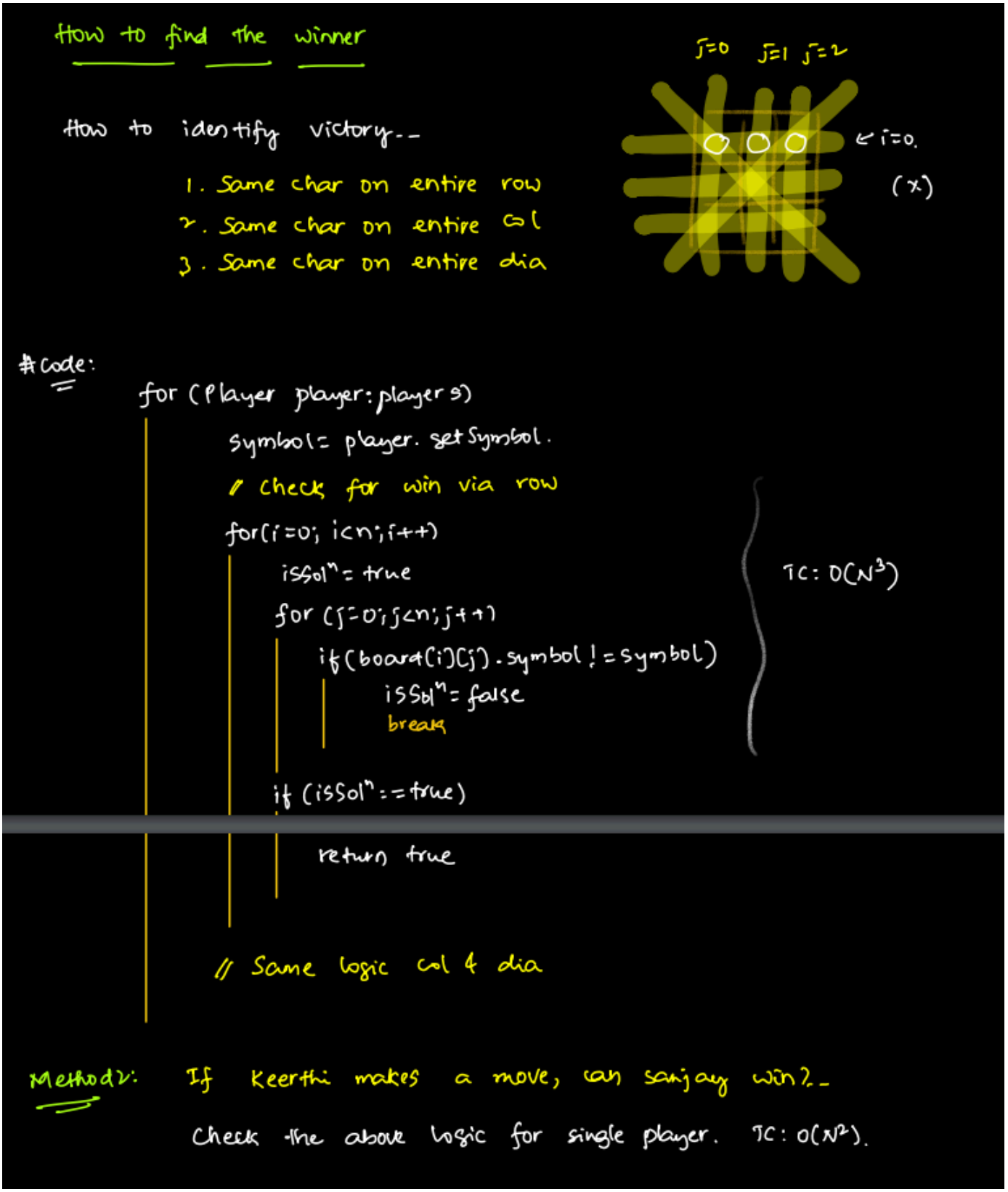
https://youtu.be/hTtO_sOW-dI

Like | Comment | Subscribe

==========

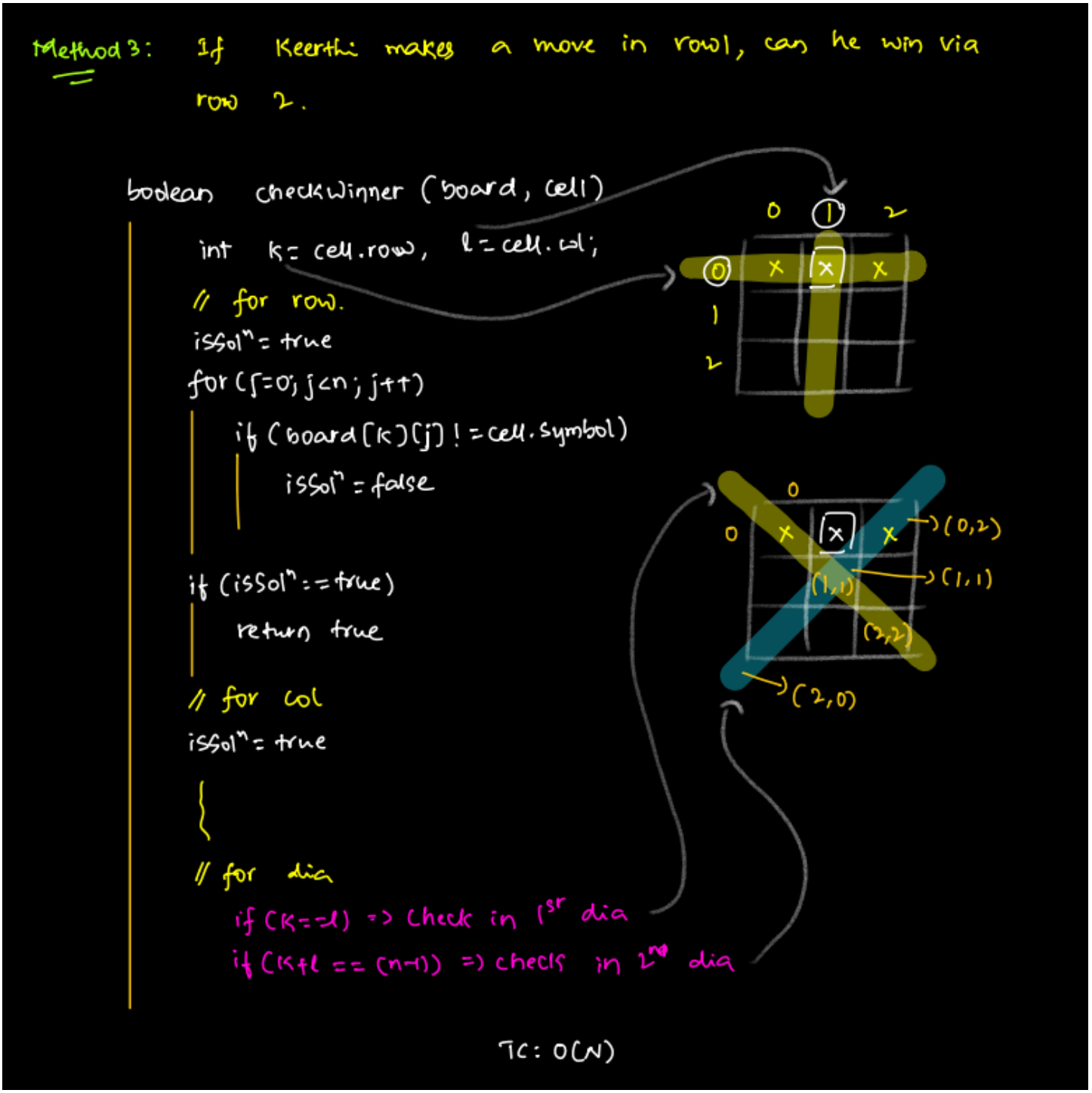
How to find the winner

If we have a grid (matrix), and we spot the same symbol in a row, column, or diagonal, that means we've found the winner but it will take **O(N^3)** complexity can we make it **O(N^2).**



We can make it **O(N^2)** just check the above logic for single player and we can reduce one loop because at time only 1 player can win.
To reduce the complexity from **O(N^2)** to **O(N)**, we can optimize the checking process. Instead of examining all rows each time, we only need to check the row, column, and diagonals associated with the current cell. For example, if we move in row 0, then row 0 remains constant while we increase the column. The same principle applies when moving in

columns; the column remains constant while we vary the row. This way, we only need to focus on the relevant elements, making the process more efficient.

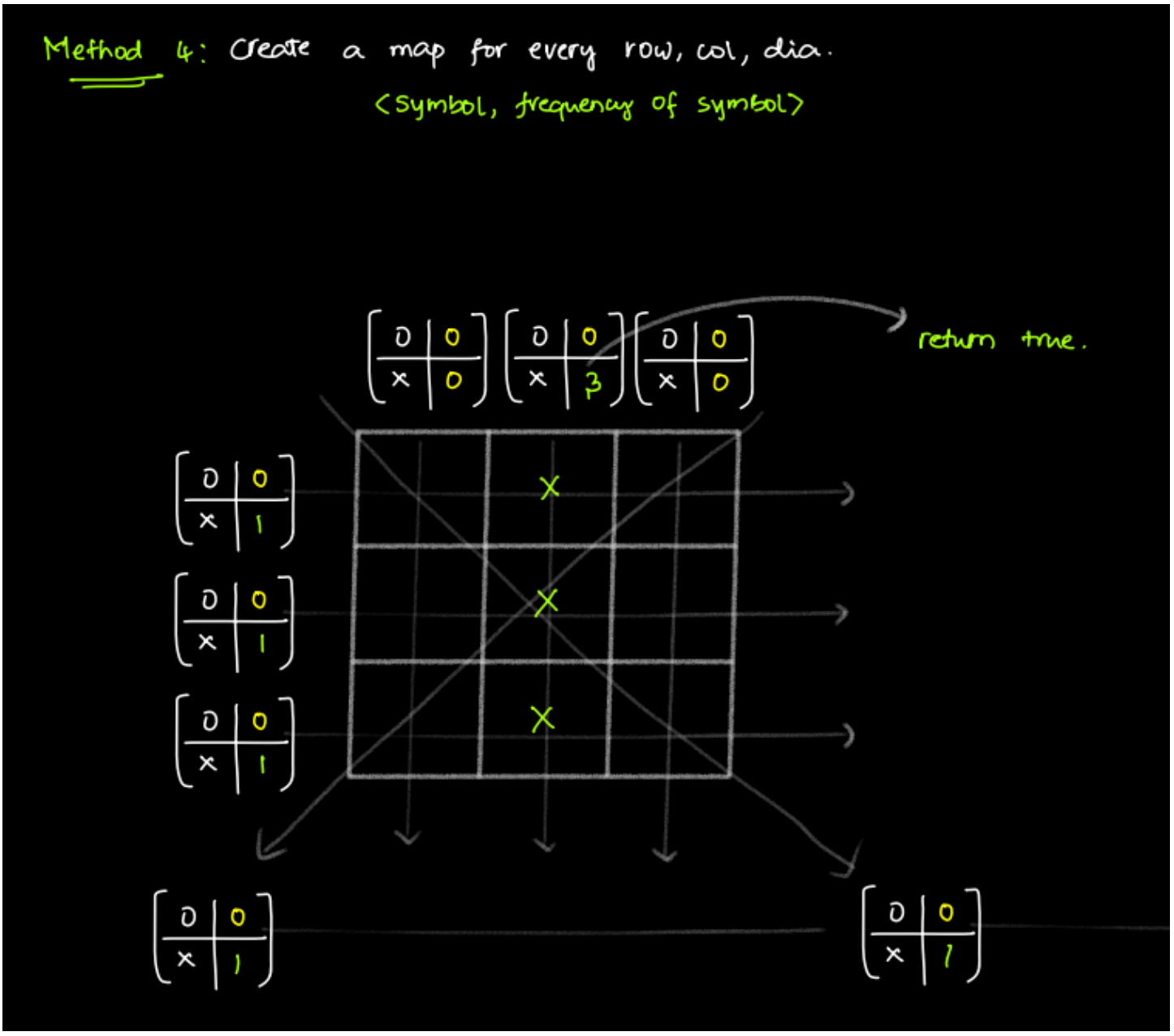


To transition from O(N) to O(1), we need to create two maps. We will store symbol as key and frequency of simbol as value. one for rows and another for columns. For each row and column, we insert and track their occurrences of all charector. Let's illustrate this with an example: if there's a cross marker in the 0th row and 1st column, we increase the count of crosses in the 0th row cell to 1. Simultaneously, in the row map, we increment the count in the 1st index cell to 1.
As we encounter additional crosses in the same column and row, we update the corresponding counts in the maps accordingly. For instance, if another cross appears just below the same column in the next row, we increase the count in the same row cell in the

map. Similarly, we increment the count for crosses in the 1st index of the column map. This process continues, keeping track of the occurrences.

Once we have completed this mapping, we can easily check if the count in any column is equal to the number of rows for the same character. If so, we return true.

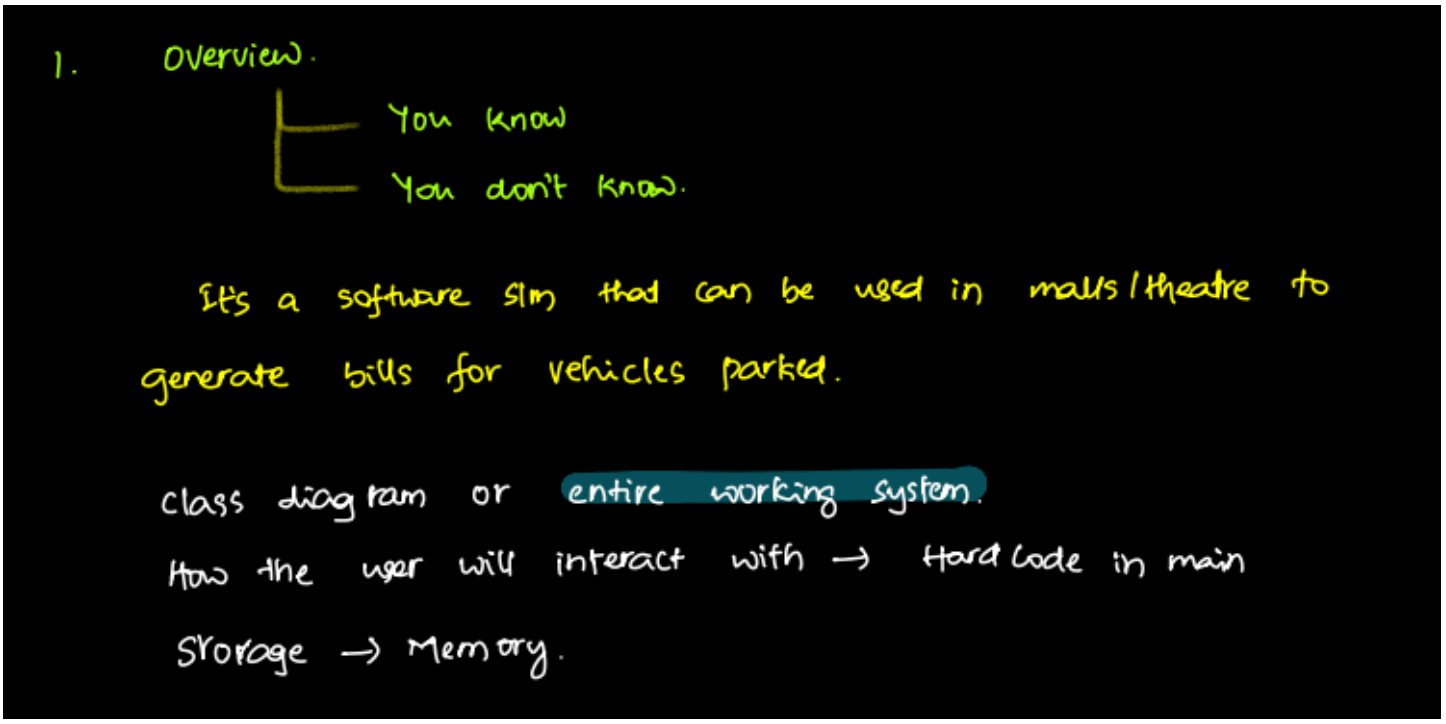
We will apply a similar mapping approach for both diagonals, monitoring the occurrence count of symbols.



Backend LLD-3: Machine Coding - 6: Design Parking Lot - Feb 15

- **Overview of parking lot**
- **Requirement gathering for parking lot**
- **Class diagram**
- **Extension : Allo partial payments**
- **Interview experience - Queue using 1 stack**

Overview of parking lot -> If we know then we will give over view and take client confirmation on our understanding and if we we do not know then we can ask to client politely for overview of requirement and then what he is expecting from us just digramme or complete solution based on his confirmation we will go ahed with quitons.



==========

Downloading a Scaler Assignment Locally: Step-by-Step Guide

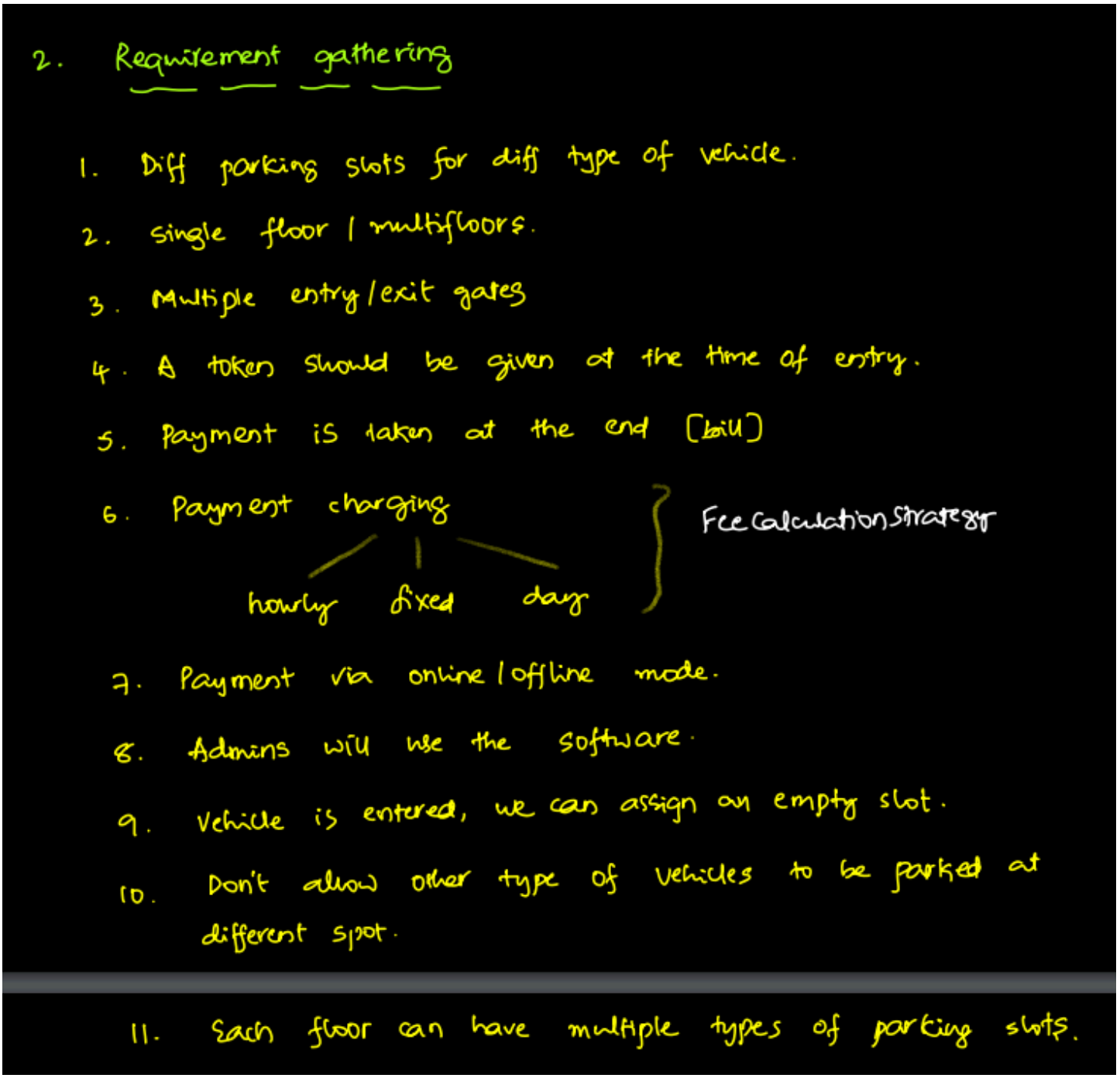
Visit

https://youtu.be/TN36hpp4he8

Like | Comment | Subscribe

==========

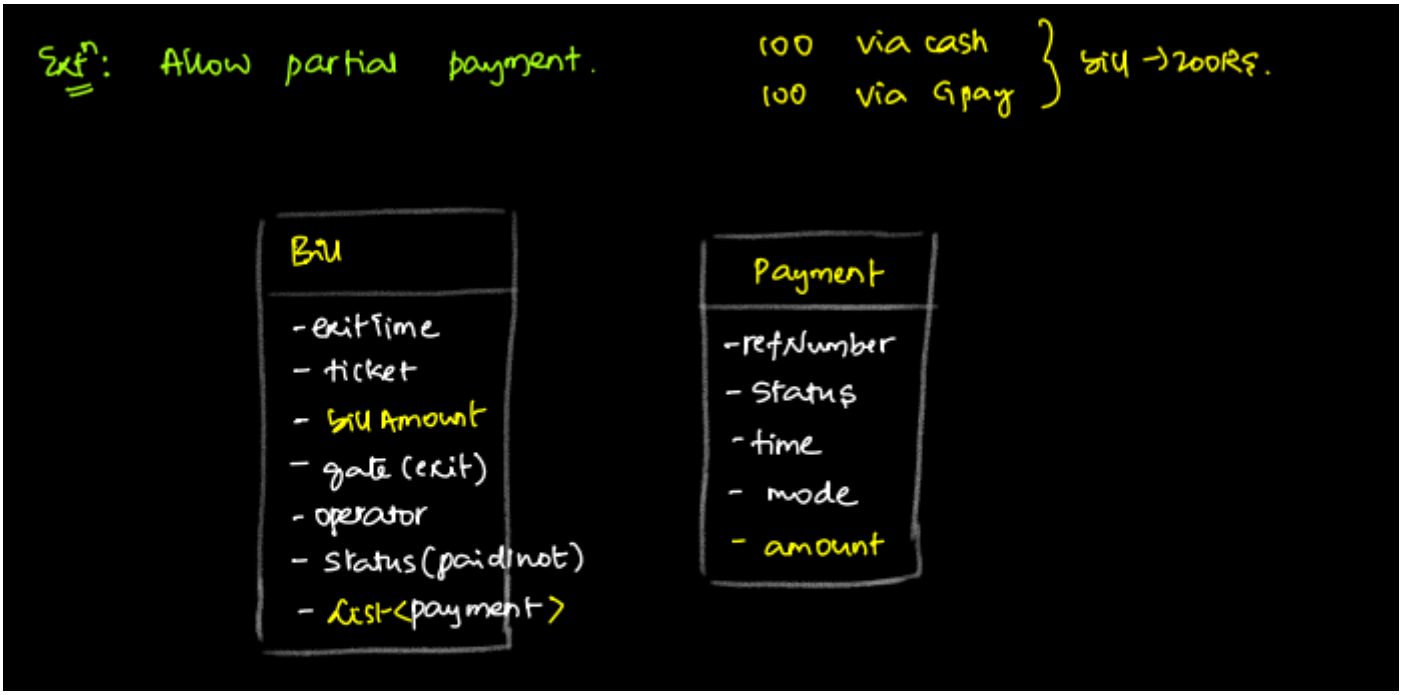
Requirement gathering for parking lot



Class Diagram ->

Extension : Allo partial payments -> Split payments are a common practice, where we provide a portion of the payment in cash and the remaining through online methods. This frequently comes up in interviews, especially if we intend to incorporate it into our existing system. To implement this, adjustments are needed in both the **bill and payment classes**.

In the bill class, where there used to be a single payment, we'll modify it to accommodate a **list of payments**. The "**amount**" field will be renamed to "**billAmount**" to better represent the total bill amount. In the payment class, we'll no longer have a field named "**amount**" since we had initially linked it to the bill amount. However, due to partial payments, each payment now needs to specify an **amount**.



Interview experience with Instructor - Queue using 1 stack -> During the interview, the task was to create a queue using only one stack. Typically, when converting a stack into a queue, two stacks are required. The process involves initially inserting all items into the first stack. When a dequeue operation is performed, all items are transferred to the second stack. The top item is then popped out, and if new items arrive, they are inserted into the empty first stack. When a dequeue operation is initiated, data can be swiftly removed from the second stack in constant time (O(1)). We will continue popping items until the second stack is empty. In the scenario where the second stack is empty and a new dequeue request is received, we refill it with the contents of the first stack and return the popped-out item. This approach is the conventional method for constructing a queue, offering an average time complexity of O(1), except during the initial dequeue operation, where all elements are shifted from the first stack to the second, resulting in a slightly higher time complexity.

However, the challenge was to implement a queue using only one stack. To achieve this, we needed to think creatively and solve the problem on paper multiple times. The key insight was to use recursion. With recursion, we could perform the task by making a recursive call for each item. The base case condition was set such that when the stack size becomes 1, the popped item is returned. This recursive approach allowed us to simulate the queue behavior using just one stack.

Queue using stacks.

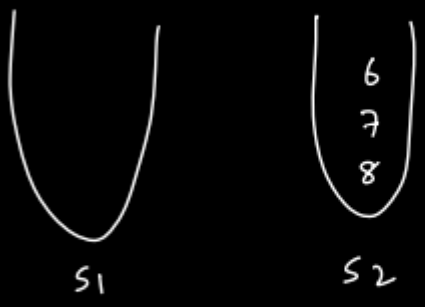
2 stacks.

1 2 3 4

1, 2, 3, 4, dq, dq, 5, 6, dq, dq,

7, 8, dq.

5

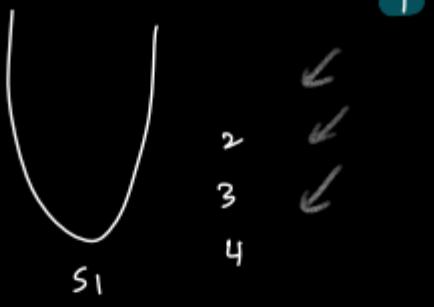


TC: enque : O(1)

deque : O(1) in average.

1 stack.

=) dq.




```
enqueue()
    push to S1.
```

```
int dequeue()
    if (S1.size()==1)
        return S1.pop()

    top = S1.pop()
    output = dequeue();
    S1.push(top);
    return output
```

Backend LLD-3: Machine Coding - 7: Code Parking Lot 1 - Feb 17

- **Cardinalities revision**
- **How to convert class diagram to schema diagram**
- **How to represent relations**
- **How to represent enums**
- **creating schema for parking lot**
    - **Step 1 : Create tables for each entit**
    - **Step 2 : Add primitive attributes of entities as cols in tables**
    - **Step 3 : Handling relations**
    - **Step 4 :Avoid recalculating duplicate relations**
- **Use of DTOs in flipkart**

Cardinalities revision -> Cardinality in the context of database schema design refers to the relationship between tables, specifically the number of instances of one entity that can be related to another entity. Cardinality is often expressed using terms like "one-to-one," "one-to-many," or "many-to-many." Understanding cardinality is crucial for designing efficient and normalized database schemas.

Here are the common cardinalities and their symbols:

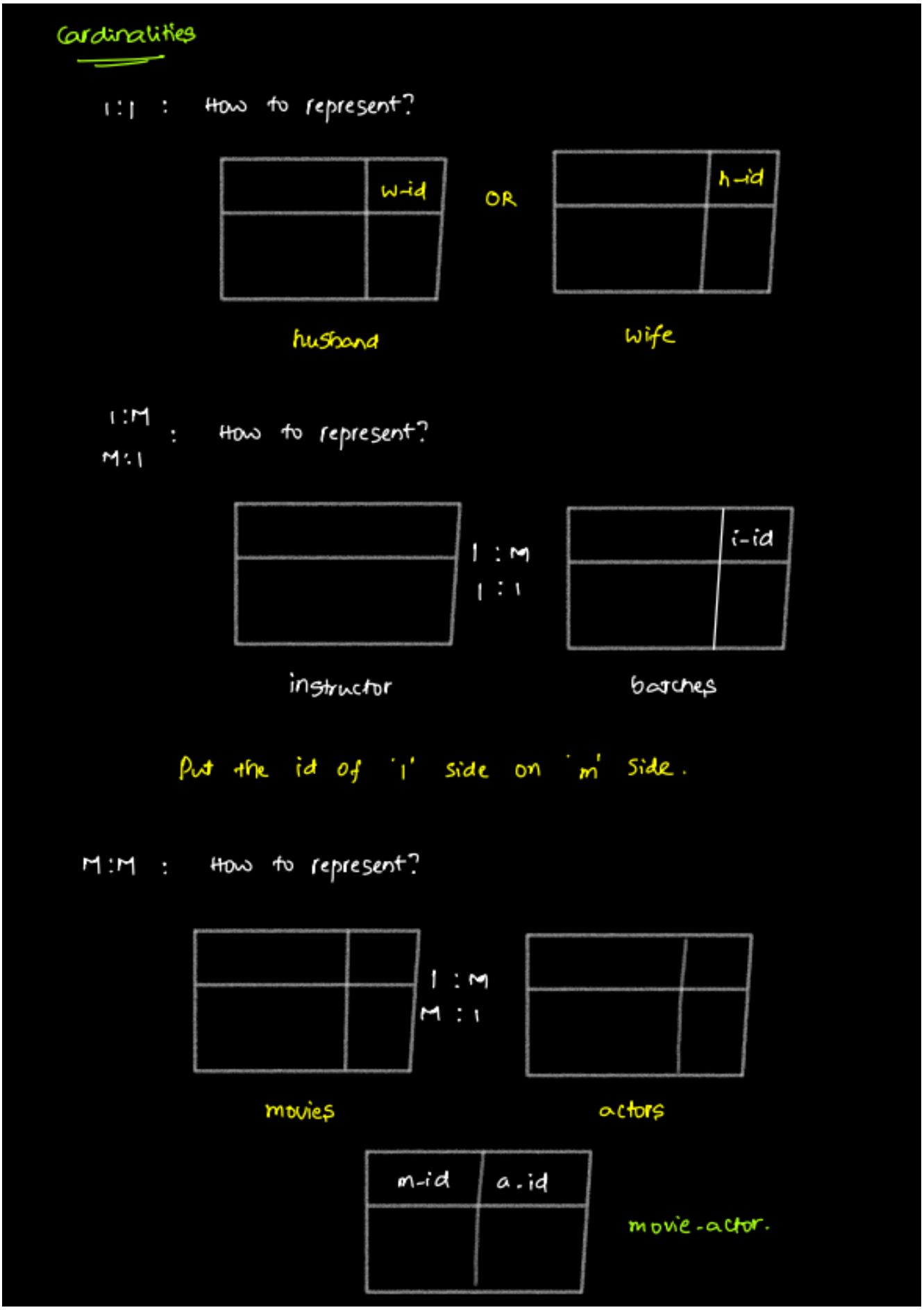
One-to-One (1:1): In a one-to-one relationship, each record in the first table is related to only one record in the second table, and vice versa.
Represented by a straight line between the two entities.
Example: Employee and EmployeeDetails tables, where each employee has one set of details.

One-to-Many (1:N): In a one-to-many relationship, a record in the first table can be related to multiple records in the second table, but each record in the second table is related to only one record in the first table.
Represented by a line with a "crow's foot" at the "many" end.
Example: Department and Employee tables, where one department has many employees.

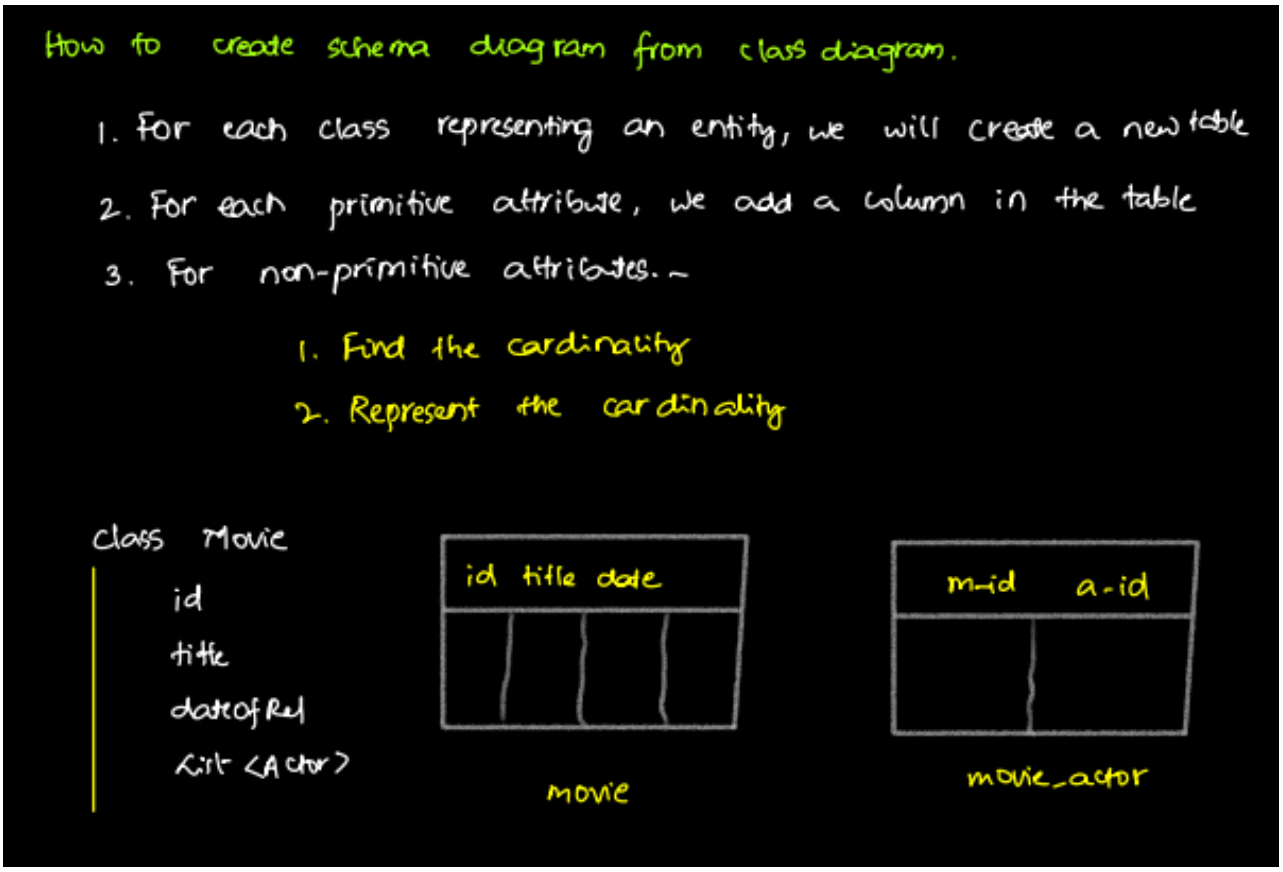
Many-to-One (N:1): This is essentially the reverse of one-to-many. Many records in the first table can be related to a single record in the second table.
Represented by a line with a "crow's foot" at the "one" end.
Example: Many products belong to one category.

Many-to-Many (N:N): In a many-to-many relationship, multiple records in the first table can be related to multiple records in the second table.
Implemented using a junction table (also known as an associative or linking table) that breaks the many-to-many relationship into two one-to-many relationships.
Represented by a line with "crow's foot" at both ends.
Example: Students and Courses tables, where each student can enroll in multiple courses, and each course can have multiple students.

When designing a database schema, it's crucial to identify and define the cardinality correctly to ensure data integrity and optimize query performance. Normalization techniques are often applied to avoid redundancy and reduce the likelihood of update anomalies in the database.



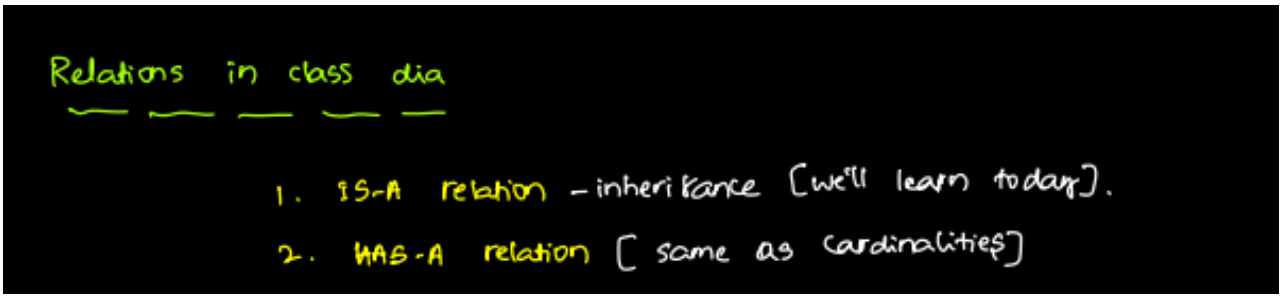
How to convert class diagram to schema diagram -> Converting a class diagram to a database schema diagram involves mapping the concepts and relationships in the class diagram to database tables, fields, and relationships. Here are the general steps to help you with the conversion process:



In the provided information, each class represents an entity (any class storing data is termed an entity, excluding subclasses).
Step 1: Examine all entities and create a table for each entity.
Step 2: Identify column names. For class columns, consider two aspects: primitive fields and non-primitive fields (reference types).

- For primitive fields, directly add them to the table.
- For non-primitive/reference type fields, determine the cardinality. Add field names based on cardinality; for example, if it's a 1:m relationship, include a reference to the "1" side in the "many" side. In cases like the movie example, where there is a List of actors representing a reference type with a many-to-many relation, create an additional mapping table. For m:m relations, always create a mapping table. However, for 1:1 or 1:m relations, simply add an ID column in the related table.
- Identify all cardinalities for reference type fields and incorporate them into the table as columns.

Relations in class diagrams ->



There are two type of relation in class

IS-A Relationship:

- Also known as inheritance or specialization.
- It represents a relationship where one class is a specialized version of another class.
- The subclass (child class) "IS-A" type of the superclass (parent class).
- It's often used to model a hierarchy and share common attributes and behavior.

```
class Animal { ... }
class Dog extends Animal { ... }
```

In this example, a Dog "IS-A" type of Animal.

HAS-A Relationship:

- Also known as composition or aggregation.
- It represents a relationship where one class has another class as a component or part.
- It is used to model a "whole-part" relationship.

```
class Car {
    Engine engine;
    // Other attributes and methods
}
```

In this example, a Car "HAS-A" relationship with an Engine, indicating that a car is composed of an engine.

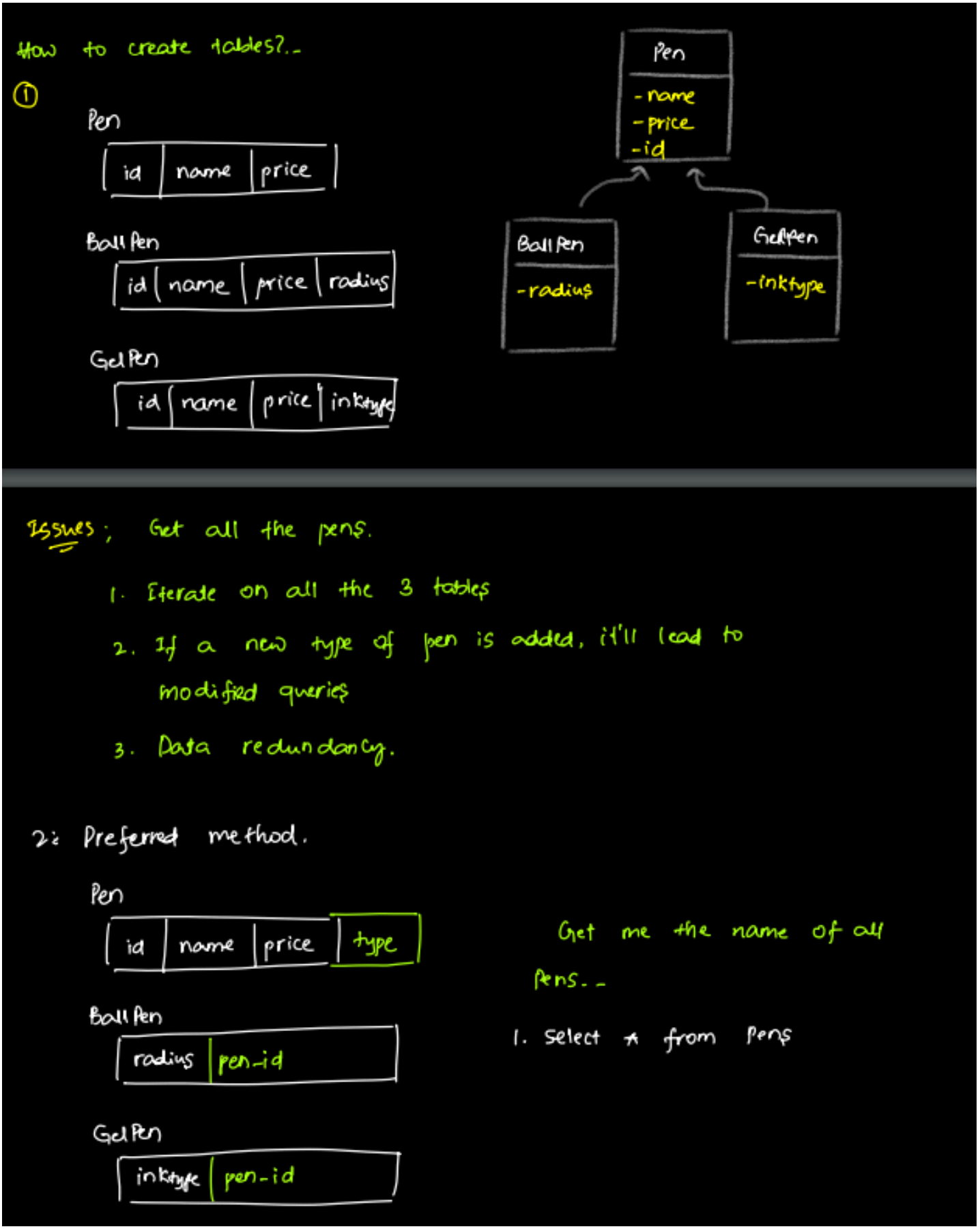
It's important to note the difference between the two:

IS-A Relationship (Inheritance):

- Implies a relationship of specialization or a kind of hierarchy.
- Typically involves a subclass inheriting attributes and behaviors from a superclass.
- Represents an "is a" or "kind of" relationship.

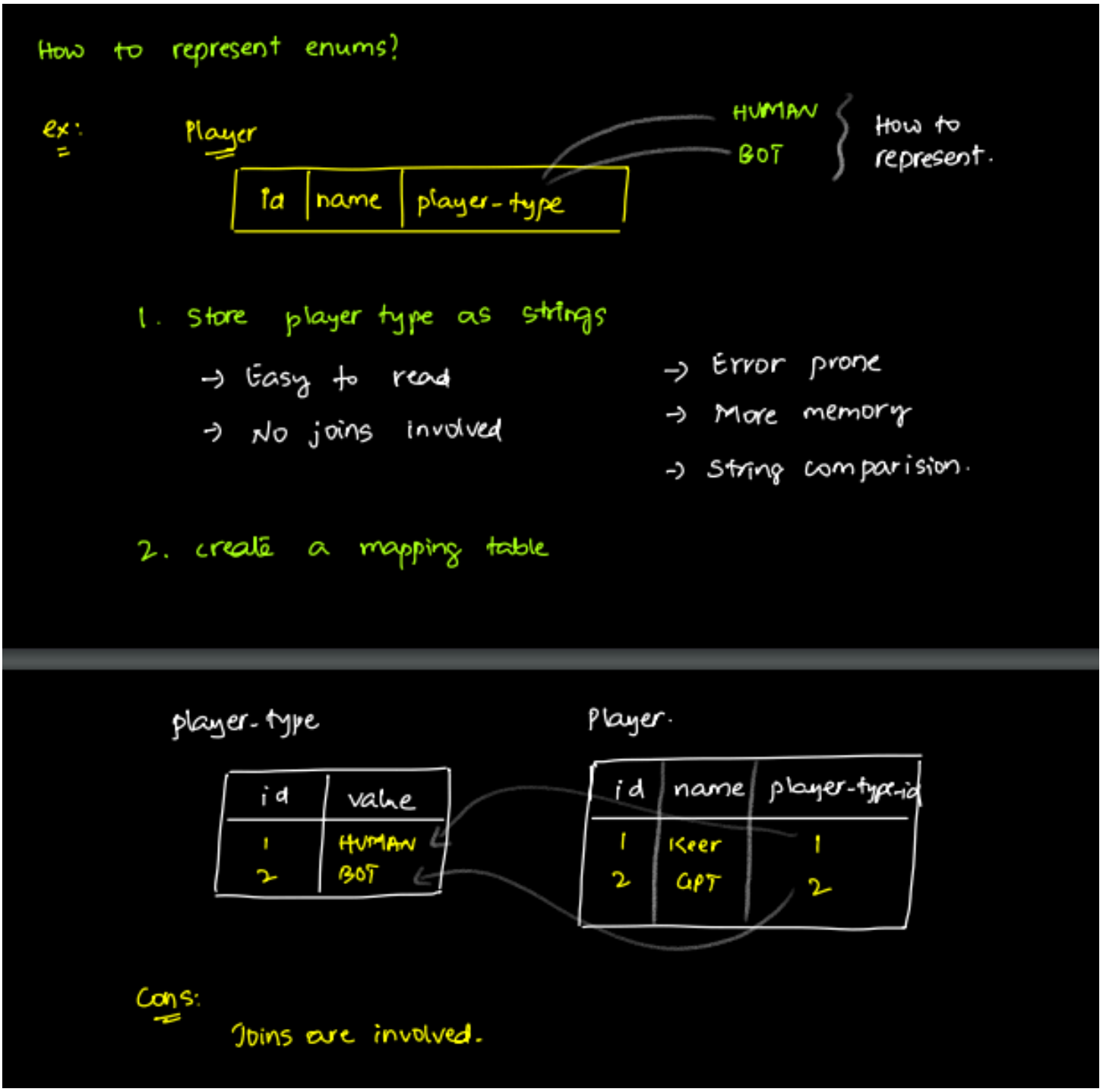
HAS-A Relationship (Composition):

- Implies a relationship of composition or aggregation.
- Typically involves one class containing an instance of another class.
- Represents a "has a" or "is composed of" relationship.

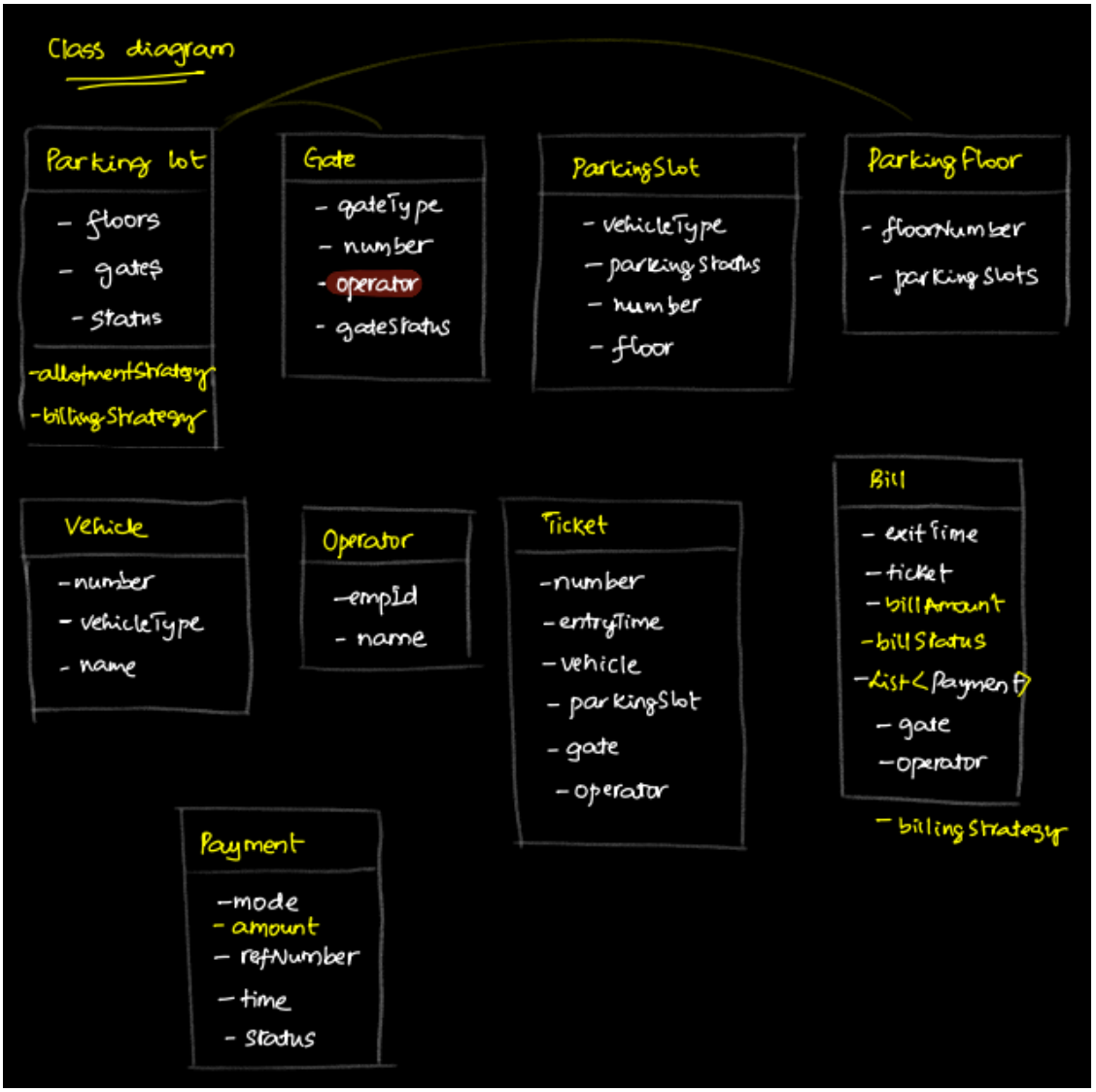


We can observe the effective utilization of relationships to address issues arising from the creation of distinct tables, such as in the case of a pen example. This approach minimizes code redundancy and enhances scalability. By employing relationships like IS-A and HAS-A and considering cardinality, we incorporate IDs to retrieve data as needed through joins.

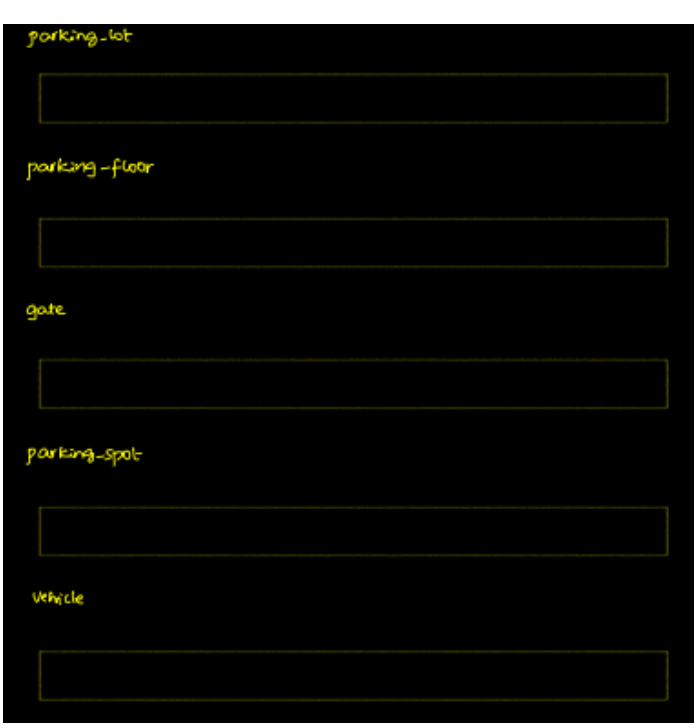
How to represent enums -> Storing strings is not a recommended approach as it introduces complexity and is error-prone. Different developers may write strings differently, leading to inconsistencies. Additionally, storing strings requires more memory, and string comparison can be more resource-intensive. A more efficient solution is to store enums by creating a mapping table and using the enum's ID as a reference. To represent an enum, a mapping table needs to be created for the ENUM, storing its ID instead of the enum itself.



Creating schema for parking lot



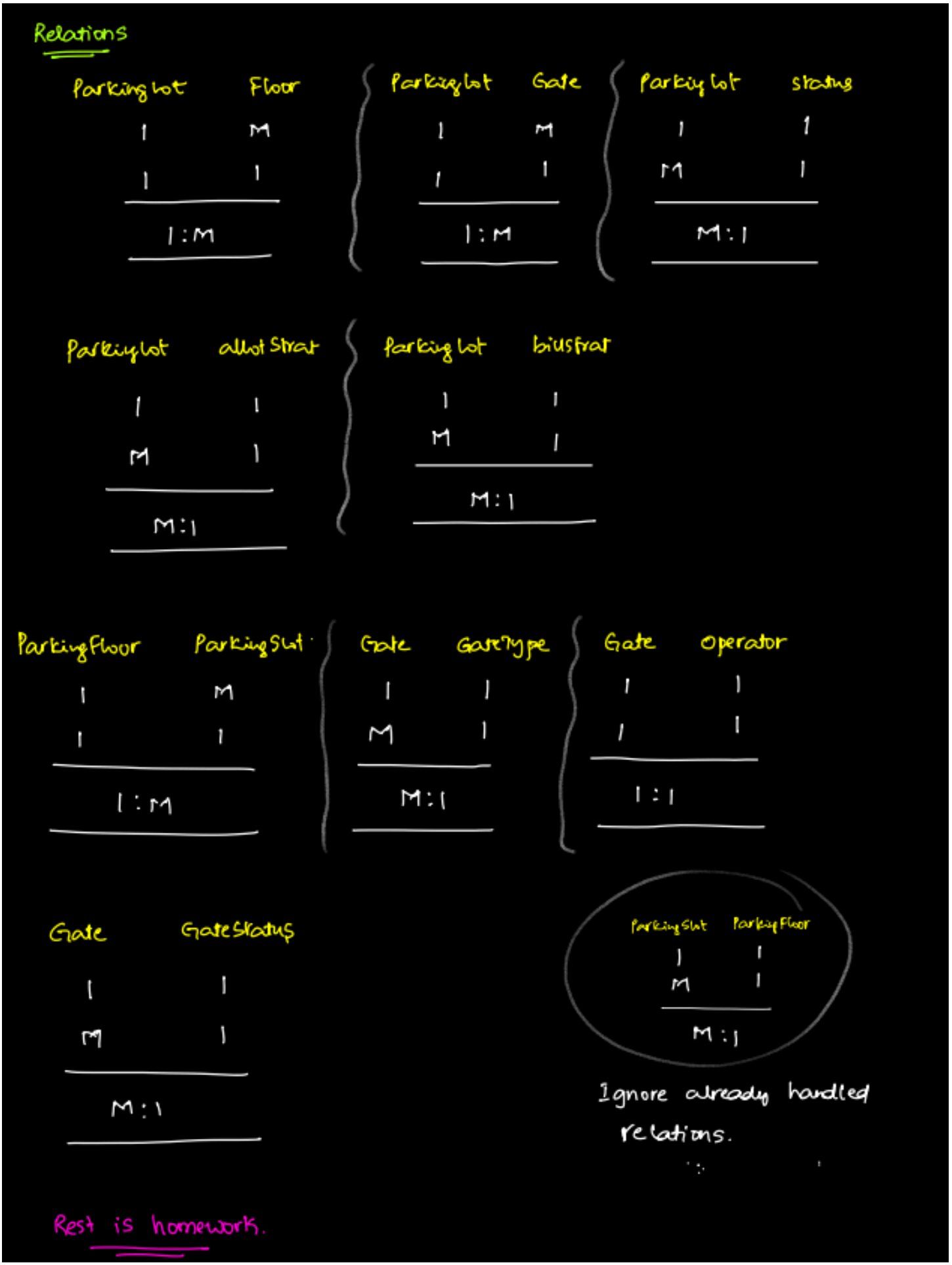
Step 1 : Create tables for each entit



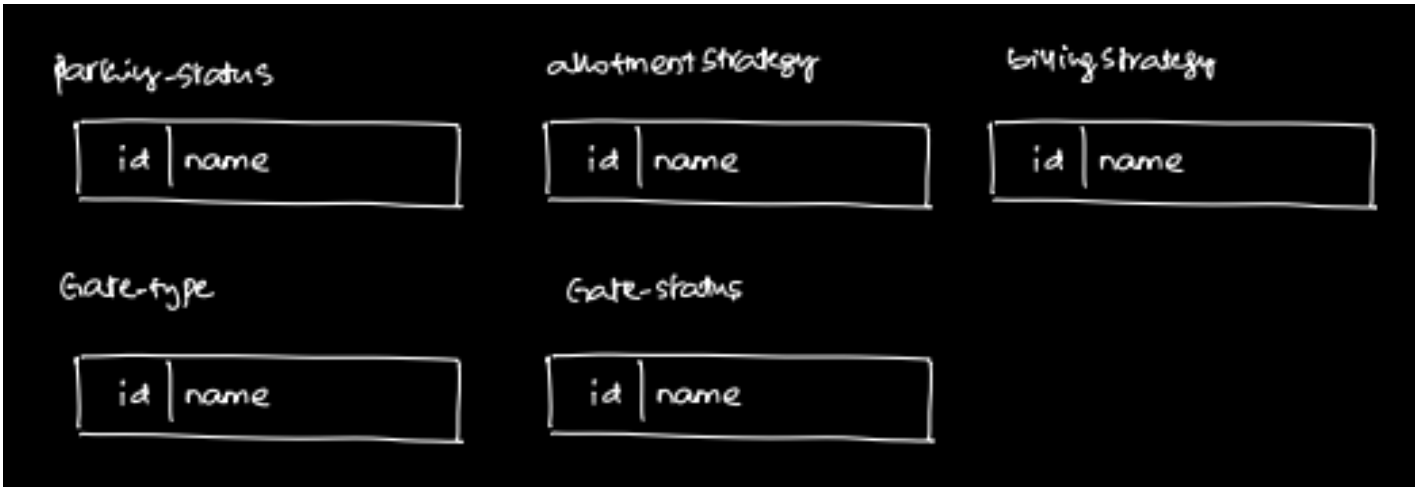
Step 2 : Add primitive attributes of entities as cols in tables

Primitives become direct column

Step 3 : Handling relations



Enums



DTOs -> DTO stands for Data Transfer Object. DTOs are objects used to transfer data between software application subsystems or layers that may have different structures. They are often employed to encapsulate data and send it across different parts of a system, especially when those parts might reside on different physical or logical tiers.

Here are some key points about DTOs:

Purpose:

- DTOs serve as a container for data that needs to be transferred between components of a system.
- They help in reducing the number of method calls and data transfer overhead, especially in distributed systems.

Structure:

- DTOs typically consist of fields (attributes or properties) to hold data. They may not have significant business logic or behavior.
- Fields in a DTO represent the data that needs to be exchanged between different parts of the system.
-

Data Transfer: DTOs facilitate the transfer of data between layers, such as between the presentation layer and the business logic layer or between the server and the client in a distributed system.

Immutability: DTOs are often designed to be immutable, meaning that their state cannot be changed once they are created. This helps in ensuring consistency during data transfer.

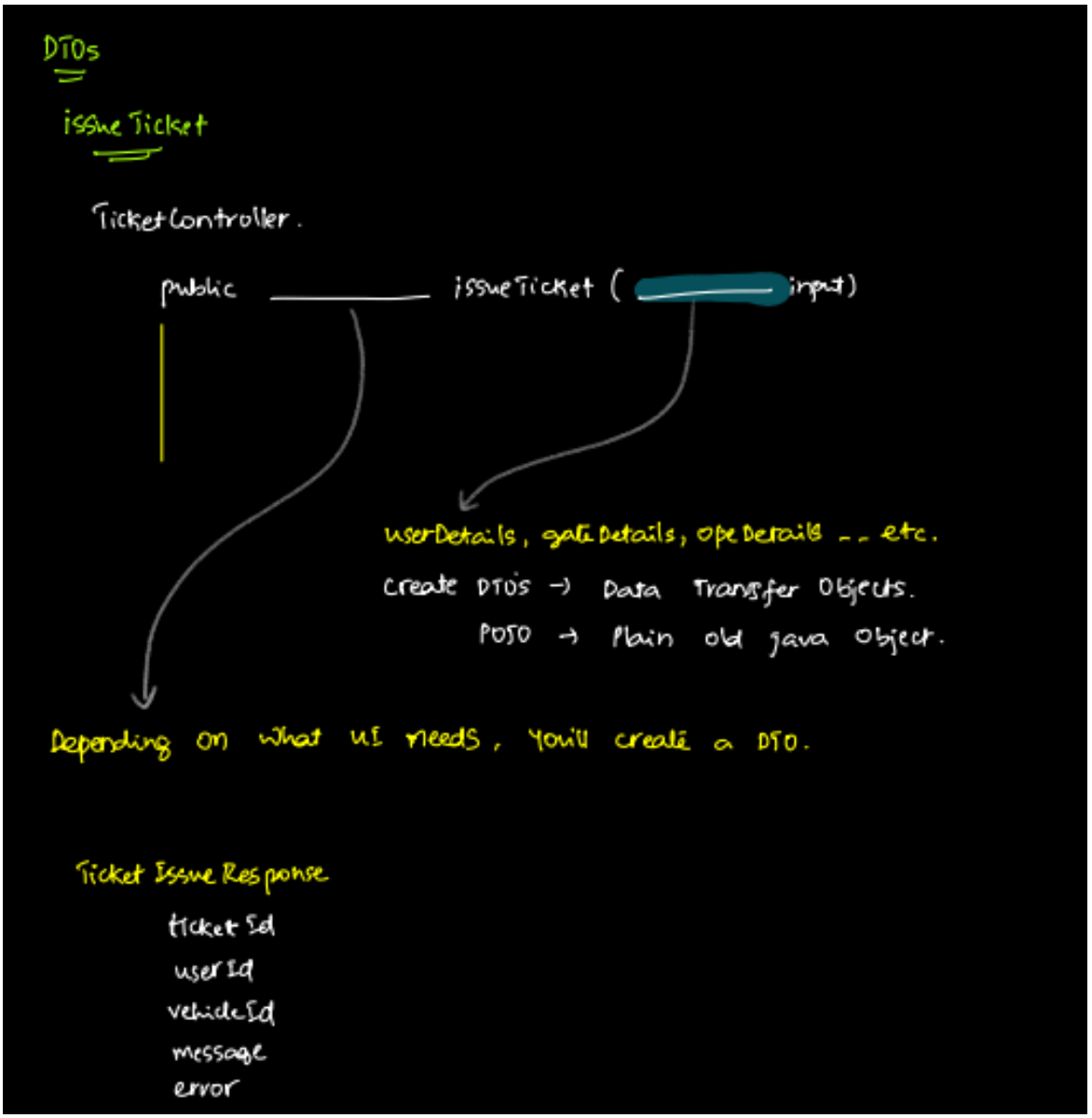
```
public class UserDTO {
    private String username;
    private String email;
    // Other fields, getters, setters, etc.
}
```

In this example, UserDTO is a simple DTO class with fields representing data that might be transferred between different layers or components.

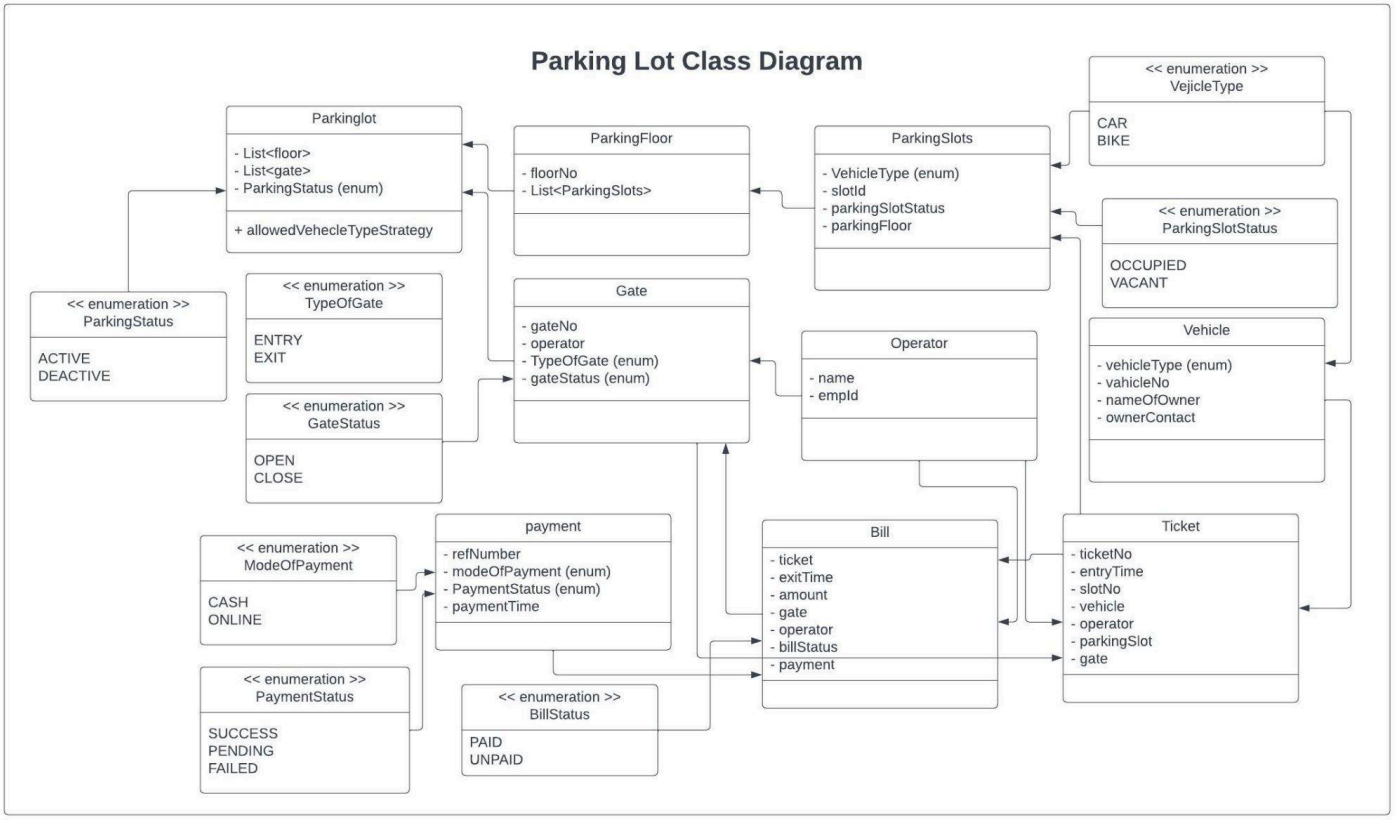
Mapping:

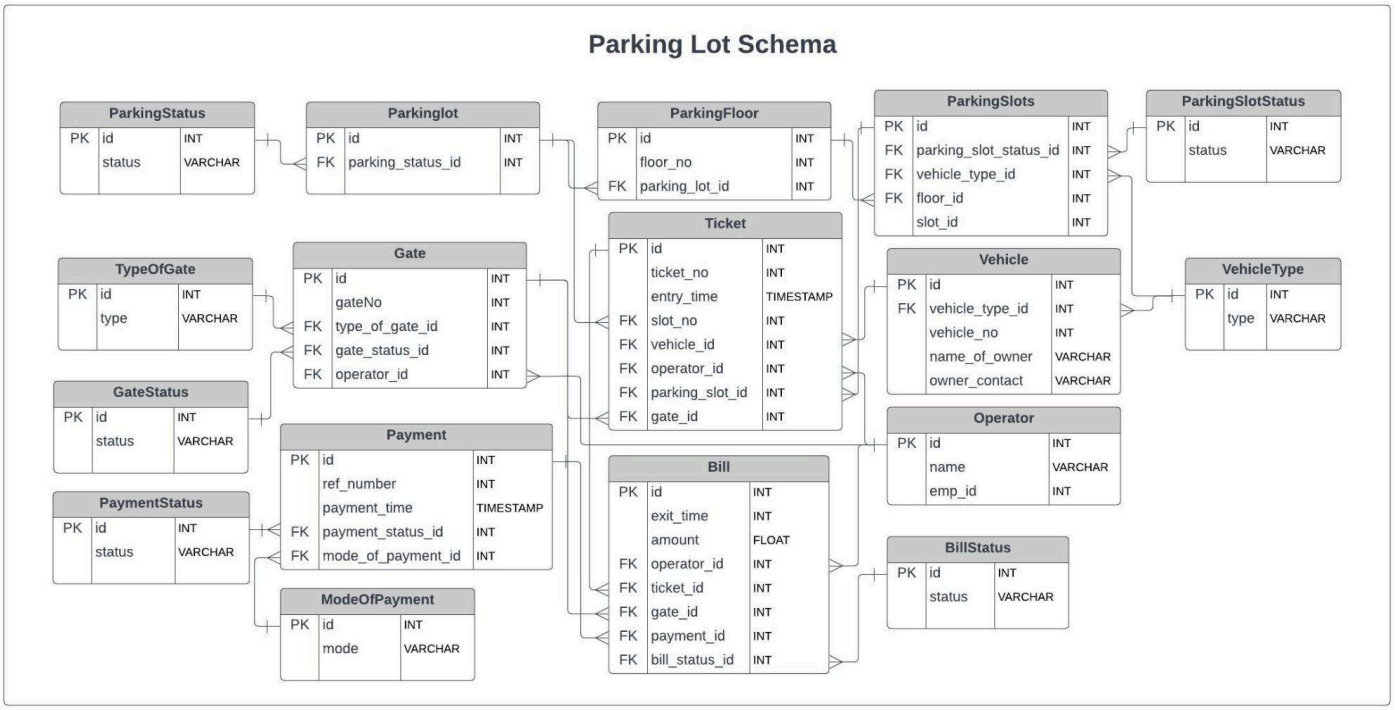
DTOs are often used in conjunction with mapping tools or manual mapping code to convert data between DTOs and the entities or objects used internally within specific layers.
DTOs are commonly employed in scenarios where the internal representation of data in different parts of the system is not suitable for direct transfer due to differences in structure, granularity, or other reasons. They help in promoting a clear separation of concerns and maintainability in software architectures.

Use of DTOs in flipkart



Backend LLD-3: Machine Coding - 8: Code Parking Lot 2 - Feb 21





Step 1: Begin by establishing a "Models" folder. Initiate the process of crafting each model within this folder, ensuring the inclusion of essential properties and the implementation of getter and setter methods as needed

1. ParkingLot

---

private Long id;
private List<Floor> floors;
private List<Gate> gates;
private ParkingLotStatus parkingLotStatus;

2. Floor

---

private int floorNumber;
private List<ParkingSpot> parkingSpots;

3. ParkingSpot

---

private int number;
private VehicleType vehicleType;
private ParkingSpotStatus status;
private Floor floor;

4. VehicleType (enum)

TWO_WHEELER, FOUR_WHEELER, HEAVY_VEHICLE

5. ParkingSpotStatus (enum)

AVAILABLE, FILLED

6. Gate

---

private Long number;
private GateType gateType;
private Operator operator;
private GateStatus gateStatus;

7. GateType (enum)

ENTRY, EXIT

8. Operator

---

private Long empId;
private String name;

9. ParkingLotStatus (enum)

ACTIVE, INACTIVE

10. Ticket

---

private String number;
private Date entryTime;
private Vehicle vehicle;
private ParkingSpot parkingSpot;
private Gate gate;
private Operator operator;

11. Vehicle

---

private String number;
private VehicleType type;
private String ownerName;

12. Bill

---

private Date exitTime;
private Ticket ticket;
private int billAmount;
private BillStatus billStatus;
private List<Payment> payments;
private Gate gate;
private Operator operator;

13. BillStatus (enum)

PAID, IN_PROGRESS, PENDING

14. Payment

---

private PaymentMode paymentMode;
private String refNumber;
private int amount;
private Date time;
private PaymentStatus paymentStatus;

15. PaymentMode (enum)

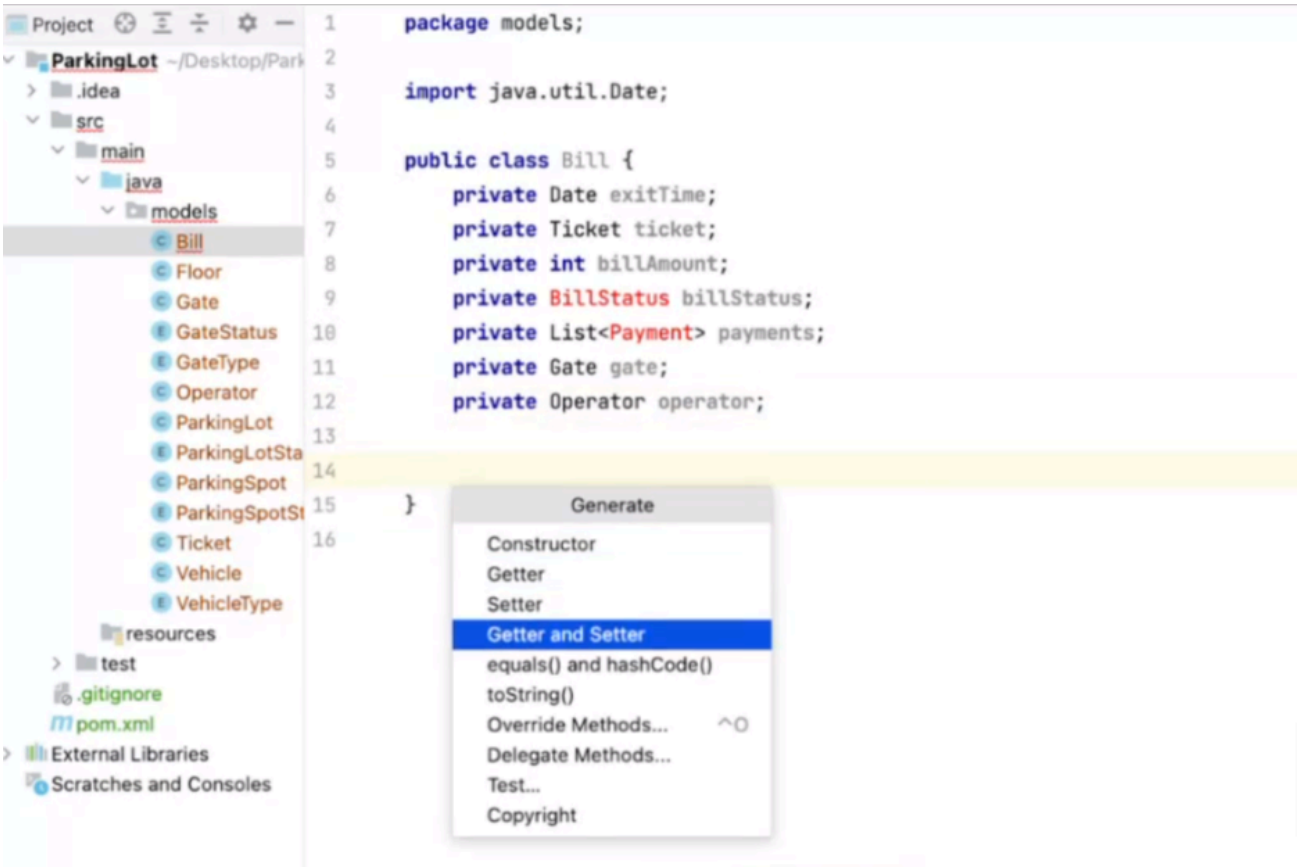
CASH, CARD, UPI, NETBANKING

16. PaymentStatus (enum)

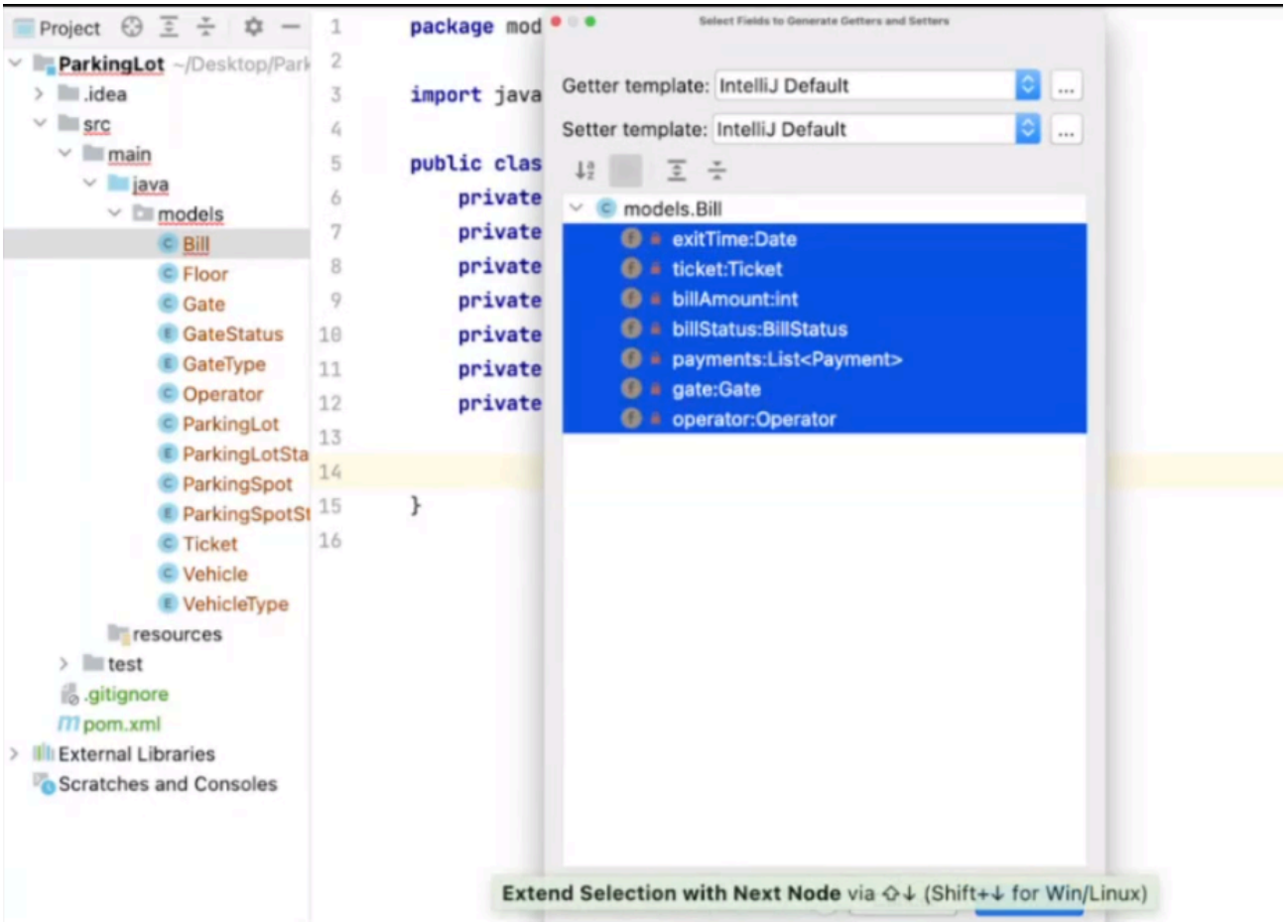
PENDING, PAID, IN_PROGRESS

Create getter and setter for all the class module like below

Right click on file and select Getter and Setter



Select all the fields and then press the "Enter" key.



Now, we won't be implementing the entire parking lot system. Instead, we will focus on developing a single feature: **generating a ticket**.

Step 2: Create the controllers package and generate TicketController.
Write the method issueTicket within TicketController, specifying the return type as IssueTicketResponse and the parameter type as IssueTicketRequest.

Now, declare TicketService within TicketController above **IssueTicketResponse**. Call the issueTicket method on the ticketService instance, passing the request parameter to encapsulate the business logic.

```
import dtos.IssueTicketRequest;
import dtos.IssueTicketResponse;
import services.TicketService;

public class TicketController {

  private TicketService ticketService;

  public IssueTicketResponse issueTicket(IssueTicketRequest request){
      return ticketService.issueTicket(request);
  }
}
```

Step3: In the dtos directory, create IssueTicketResponse class with the following fields along with getter and setter methods: Create a constructor with all the fields, ensuring to include the Ticket object, as we need this in the TicketController class.

```
public class IssueTicketResponse {
  private Ticket ticket;
  private String response;
  private String message;

  public IssueTicketResponse(Ticket ticket, String response, String message) {
      this.ticket = ticket;
      this.response = response;
      this.message = message;
  }

  // Getter and setter methods
}
```

Step4: create **IssueTicketRequest** class in the dtos package with the following fields and generate all getter and setter methods:

```
public class IssueTicketRequest {
  private VehicleType vehicleType;
  private String vehicleNumber;
  private String owner;
  private Long gateId;
  private Long parkingLotId;

  // Getter and setter methods
}
```

These classes and methods will facilitate the issuance of tickets in your parking lot system.

Step 5: Create a Services package in the java folder, then generate the **TicketService** class with the issueTicket method, which will be invoked by TicketController:

```
package Services;
import dtos.IssueTicketRequest;
import models.Ticket;

public class TicketService {
  public Ticket issueTicket(IssueTicketRequest request){
      return null;
  }
}
```

Now, return back to **TicketController** class, and utilize the **ticketService** method response by assigning it to the Ticket object. Create a new instance of **IssueTicketResponse** and pass the required parameters, then return it:

```
import dtos.IssueTicketRequest;
import dtos.IssueTicketResponse;
import models.Ticket;
import Services.TicketService;

public class TicketController {
  private TicketService ticketService;
  public IssueTicketResponse issueTicket(IssueTicketRequest request){

      Ticket ticket = ticketService.issueTicket(request);
      return new IssueTicketResponse(ticket, "Success", "Ticket Generated");
  }
}
```

These modifications integrate the TicketService class into the TicketController, allowing the issueTicket method in the controller to utilize the response from the service and create a new IssueTicketResponse object for the ticket issuance operation.

Step 6: Come back to ticketService and implment method issueTicket and follow below steps

1. SET TIME
2. GET VEHICLE AND GATE DETAILS
3. GET PARKING LOT
4. ASSIGN AN EMPTY SPOT FOR THIS VEHICLE
5. GENERATE TICKET NUMBER
6. SAVE AND RETURN THE TICKET

1. Set the Time: Create a ticket and use a method to set the entry time.

```
public Ticket issueTicket(IssueTicketRequest request){
   Ticket ticket = new Ticket();
   ticket.setEntryTime(new Date());
}
```

2. Get Vehicle and Gate Details:

To find gate details using the gate ID, normally, we use a database. However, in this case, we'll store it in memory.
Create a new package named **repositories** and a class called **GateRepository** inside it. This class should handle basic operations on gates (create, read, update, delete).
Implement a method called **findGateById** in GateRepository that takes a gate ID, checks if it exists in a collection (HashMap), and returns the gate. If not, throw a GateNotFoundException.

```
   public class GateRepository {
   private Map<Long, Gate> gates = new HashMap<>();

   public Gate findGateById(Long id) throws GateNotFoundException {
       if (gates.containsKey(id)) {
           return gates.get(id);
       }
       throw new GateNotFoundException();
   }
}
```

Exception Handling:

Create a package named **exceptions** outside the module.
In this package, create a class called **GateNotFoundException** that extends Throwable.

```
public class GateNotFoundException {


}
```

These steps guide you through setting the entry time for a ticket, handling gate details (stored in memory), and creating an exception for when a gate is not found.

Return to the ticketService and obtain gate details from the **GateRepository**. Establish an instance of GateRepository within ticketService.

```
private GateRepository gateRepository;
```

After setting the entry time for the ticket in the **issueTicket** method, retrieve gate details:

```
Gate gate = gateRepository.findGateById(request.getGateId());
```

If an exception occurs, throw it with the method signature. Handle this exception in **TicketCounter** using a try-catch block. Return an **IssueTicketResponse** as null with a failure message.

```
public IssueTicketResponse issueTicket(IssueTicketRequest request){
  Ticket ticket = null;
  try {
      ticket = ticketService.issueTicket(request);
  } catch (GateNotFoundException e) {
      System.out.println("Gate not found ");
      return new IssueTicketResponse(null, "FAILURE", "Invalid Gate");
  }
}
```

After throwing the exception, return to **ticketService**, and following the retrieval from the repository, set the gate for the ticket.

```
Gate gate = gateRepository.findGateById(request.getGateId());
ticket.setGate(gate);
```

Make sure to include this line after obtaining the **gate** from the repository.

Now that we have the gate details, we also need the **vehicle** details. For this purpose, we'll create a **VehicleRepository** similar to the **GateRepository**. The repository will use a map with a string key and a Vehicle value, providing a **getVehicleByNumber** method.

```
public class VehicleRepository {
  private Map<String, Vehicle> vehicleMap = new HashMap<>();

  public Vehicle getVehicleByNumber(String vehicleNumber){
      return vehicleMap.get(vehicleNumber);
  }
}
```

In this case, if the vehicle is not found, **we won't throw an exception**; instead, we'll return null to the TicketService. The TicketService will then create a new vehicle if it's null.

Now, go back to the **TicketService** and declare an instance of **VehicleRepository**. After obtaining the gate and before setting the entry time, get the vehicle and set it for the ticket.

```
private VehicleRepository vehicleRepository;

// ... (other declarations)

public IssueTicketResponse issueTicket(IssueTicketRequest request) {
  // ... (other code)

  Gate gate = gateRepository.findGateById(request.getGateId());
  ticket.setGate(gate);

  Vehicle vehicle = vehicleRepository.getVehicleByNumber(request.getVehicleNumber());

  if (vehicle == null) {
```

```
        vehicle = new Vehicle(request.getVehicleNumber(),
                request.getVehicleType(), request.getOwner());

        vehicleRepository.save(vehicle);
    }

    ticket.setVehicle(vehicle);

    // ... (remaining code)
}

Also, don't forget to go back to VehicleRepository and create a save method.

public class VehicleRepository {
    private Map<String, Vehicle> vehicleMap = new HashMap<>();

    // ... (other methods)

    public void save(Vehicle vehicle){
        vehicleMap.put(vehicle.getNumber(), vehicle);
    }
}
```

These changes ensure that the TicketService has access to both gate and vehicle details, creating a new vehicle if necessary and saving it in the repository.

3. GET PARKING LOT

To obtain the parking lot information, we introduce a new repository named **ParkingLotRepository**. This repository utilizes a Map with keys of type Long and values of type ParkingLot. The method **getParkingLotById** is implemented to retrieve a ParkingLot based on the provided identifier.

```
import java.util.HashMap;
import java.util.Map;

public class ParkingLotRepository {

    private Map<Long, ParkingLot> parkingLotMap = new HashMap<>();

    public Map<Long, ParkingLot> getParkingLotMap() {
        return parkingLotMap;
    }

    public void setParkingLotMap(Map<Long, ParkingLot> parkingLotMap) {
        this.parkingLotMap = parkingLotMap;
    }

    public ParkingLot getParkingLotById(Long id) throws ParkingLotNotFoundException {
        if (parkingLotMap.containsKey(id)) {
            return parkingLotMap.get(id);
        }
        throw new ParkingLotNotFoundException();
    }
}
```

In case a parking lot is not available, we extend the method signature to include the ParkingLotNotFoundException and create a new exception class, ParkingLotNotFoundException, within the exceptions package.

```
package exceptions;
public class ParkingLotNotFoundException extends Throwable {
}
```

Returning to the **TicketService**, we declare the **ParkingLotRepository** as a dependency.

```
 private final ParkingLotRepository parkingLotRepository;

 public TicketService(ParkingLotRepository parkingLotRepository) {
   this.parkingLotRepository = parkingLotRepository;
 }
```

After setting the vehicle, we retrieve the corresponding parking lot using the provided ParkingLotId.

```
ParkingLot parkingLot= parkingLotRepository.getParkingLotById(request.getParkingLotId());
```

We then update the **issueTicket** method to handle the newly introduced **ParkingLotNotFoundException** and proceed to the **TicketController** to manage this error.

```
 try {
 ticket = ticketService.issueTicket(request);
 } catch (GateNotFoundException e) {
       System.out.println("Gate not found ");
   return new IssueTicketResponse(null, "FAILURE", "Invalid Gate");
 } catch (ParkingLotNotFoundException e) {
       System.out.println("Parking Lot not found ");
   return new IssueTicketResponse(null, "FAILURE", "Invalid ParkingLot");
 }
```

This ensures that the application can appropriately respond to situations where a requested parking lot is not found.

Next step -> Assign an empty spot for this vehicle.

—----------

create parkingSlotStrategy

After establishing the parking lot, the next step is to generate a ticket. However, before generating the ticket, it is essential to allocate a parking spot. This process is critical because an empty space is required for ticket generation. To determine the parking spot, various factors need consideration, such as whether to assign a spot near the stairs or close to the exit gate. To address this, we will introduce a **ParkingPlaceAllotmentStrategy** interface.

Let's organize this by creating a "strategies" package and a new file for the **ParkingPlaceAllotmentStrategy** interface. This interface will define a method named

getParkingSpot:

```
// strategies/ParkingPlaceAllotmentStrategy.java
public interface ParkingPlaceAllotmentStrategy {
   ParkingSpot getParkingSpot(VehicleType vehicleType, ParkingLot parkingLot) throws
ParkingLotFullException;
}
```

Now, to implement this strategy, we will create a class called **SimpleParkingSpotAllotmentStrategy:**

```
 // strategies/SimpleParkingSpotAllotmentStrategy.java

 public class SimpleParkingSpotAllotmentStrategy implements ParkingPlaceAllotmentStrategy
 {

   @Override
   public ParkingSpot getParkingSpot(VehicleType vehicleType, ParkingLot parkingLot)
                                                    throws ParkingLotFullException {
       /*
         1. Get all the floors of the parking lot For each floor, check the slots with
            the specified vehicle type and a status of being free
            Return the parking spot.
        */

       return null;
   }
 }
```

In this class, the **getParkingSpot** method outlines the strategy for allocating parking spots. It involves iterating through each floor of the parking lot, checking slots based on the specified vehicle type and a free status, and ultimately returning the chosen parking spot.

Next step come back to ticketService
Create a instance of **ParkingPlaceAllotmentStrategy parkingPlaceAllotmentStrategy**
The next step is deciding where to store the strategy for allocating parking spots. We have two options: keeping it with the vehicle or storing it in the ticket. We'll choose to store it in the ticket for simplicity.

Now, go to the **Ticket** class and create a space to hold the parking allocation strategy. To do this, declare an instance of ParkingPlaceAllotmentStrategy and set up a method to assign it:

```
public class Ticket {
  private ParkingPlaceAllotmentStrategy parkingPlaceAllotmentStrategy;

  public void setParkingPlaceAllotmentStrategy(ParkingPlaceAllotmentStrategy
                                               parkingPlaceAllotmentStrategy) {
      this.parkingPlaceAllotmentStrategy = parkingPlaceAllotmentStrategy;
  }
  // Other ticket-related methods...
}
```

In addition, we need this information when issuing a ticket. To achieve this, create another instance of **parkingPlaceAllotmentStrategy** in the **IssueTicketRequest** DTO and include a method to retrieve it:

```
public class IssueTicketRequest {
  private ParkingPlaceAllotmentStrategy parkingPlaceAllotmentStrategy;
  public ParkingPlaceAllotmentStrategy getParkingPlaceAllotmentStrategy() {
      return parkingPlaceAllotmentStrategy;
  }
  // Other request-related methods...
}
```

After setting up this instance, move on to the **TicketService** class. Fetch the parkingPlaceAllotmentStrategy after obtaining the parkingLot:

```
public class TicketService {
  // Other methods...
  ParkingPlaceAllotmentStrategy allotmentStrategy =
                                          request.getParkingPlaceAllotmentStrategy();
  ParkingSpot parkingSpot =
          allotmentStrategy.getParkingSpot(request.getVehicleType(), parkingLot);
      ticket.setParkingSpot(parkingSpot);
      ticket.setNumber(request.getVehicleNumber()+ UUID.randomUUID());
  // Other ticket-related actions...
}}
```

Once you have the **parkingLot** and **allotmentStrategy**, use the **getParkingSpot** method to determine the **parkingSpot**. Finally, set the parkingSpot and ticket number (convert from long to string) using the vehicle number and UUID in the same **TicketService** class.

The next step is to create a ticket, and for that, we need a repository. So, we'll make a new repository called **TicketRepository**.

```
public class TicketRepository {

  private Map<String, Ticket> ticketMap = new HashMap<>();

  public Map<String, Ticket> getTicketMap() {
      return ticketMap;
  }

  public void setTicketMap(Map<String, Ticket> ticketMap) {
      this.ticketMap = ticketMap;
  }

  public void save(Ticket ticket){
      ticketMap.put(ticket.getNumber(), ticket);
  }
}
```

After creating the **TicketRepository**, we go back to the **TicketService**. Here, we save the ticket in the repository and then return the ticket.

```
public class TicketService {
  private GateRepository gateRepository;
  private VehicleRepository vehicleRepository;
  private ParkingLotRepository parkingLotRepository;
  private TicketRepository ticketRepository;

  public TicketService(GateRepository gateRepository, VehicleRepository
         vehicleRepository,   ParkingLotRepository parkingLotRepository,
         TicketRepository ticketRepository) {
      this.gateRepository = gateRepository;
      this.vehicleRepository = vehicleRepository;
      this.parkingLotRepository = parkingLotRepository;
      this.ticketRepository = ticketRepository;
  }

  public Ticket issueTicket(IssueTicketRequest request) throws GateNotFoundException,
          ParkingLotNotFoundException, ParkingLotFullException {

       /*
          1. SET TIME
          2. GET VEHICLE AND GATE DETAILS
          3. GET PARKING LOT
             Assign an empty spot for this vehicle.
          4. GENERATE TICKET NUMBER
          5. SAVE AND RETURN THE TICKET
       */

      Ticket ticket = new Ticket();
      ticket.setEntryTime(new Date());

      Gate gate = gateRepository.findGateById(request.getGateId());
      ticket.setGate(gate);

   Vehicle vehicle = vehicleRepository.getVehicleByNumber(request.getVehicleNumber());
      if(vehicle==null){
          vehicle = new Vehicle(request.getVehicleNumber(), request.getVehicleType(),
          request.getOwner());
          vehicleRepository.save(vehicle);
      }

      ticket.setVehicle(vehicle);
      ParkingLot parkingLot =
                      parkingLotRepository.getParkingLotById(request.getParkingLotId());

      ParkingPlaceAllotmentStrategy allotmentStrategy =
                                request.getParkingPlaceAllotmentStrategy();

      ParkingSpot parkingSpot =
              allotmentStrategy.getParkingSpot(request.getVehicleType(), parkingLot);

      ticket.setParkingSpot(parkingSpot);
      ticket.setNumber(request.getVehicleNumber()+ UUID.randomUUID());
      ticketRepository.save(ticket);
      return ticket;
  } }
```

In simple terms, we create a repository to store tickets. The TicketService, responsible for issuing tickets, saves the generated ticket in this repository and returns the ticket to the user.

The next step is to create an App class in Controller package where we'll provide all the necessary data to set up a parking lot.

To issue a ticket, we'll use the TicketController. So, we create an instance of it:

TicketController ticketController = new TicketController();

To create a TicketController, we need the TicketService, so inside TicketController, we create a constructor and assign the TicketService:

```
private TicketService ticketService;
public TicketController(TicketService ticketService) {
  this.ticketService = ticketService;
}
```

Returning to the App class, we pass the **TicketService** to the **TicketController**. However, to create the **TicketService**, we need other objects like **GateRepository**, **VehicleRepository**, **ParkingLotRepository**, and **TicketRepository**. In the **TicketService**, we tie these with the constructor:

```
public TicketService(GateRepository gateRepository, VehicleRepository vehicleRepository,
        ParkingLotRepository parkingLotRepository, TicketRepository ticketRepository) {
  this.gateRepository = gateRepository;
  this.vehicleRepository = vehicleRepository;
  this.parkingLotRepository = parkingLotRepository;
  this.ticketRepository = ticketRepository;
}
```

Now, go back to the App class and pass the **TicketService** to the TicketController:

```
TicketService ticketService = new TicketService(gateRepository, vehicleRepository,
   parkingLotRepository, ticketRepository);
TicketController ticketController = new TicketController(ticketService);
```

To create the **TicketService**, we have to instantiate all the required instances of **GateRepository, VehicleRepository, ParkingLotRepository**, and **TicketRepository**:

```
VehicleRepository vehicleRepository = new VehicleRepository();
ParkingLotRepository parkingLotRepository = new ParkingLotRepository();
parkingLotRepository.getParkingLotMap().put(11, parkingLot);

TicketRepository ticketRepository = new TicketRepository();
TicketService ticketService = new TicketService(gateRepository, vehicleRepository,
            parkingLotRepository, ticketRepository);
TicketController ticketController = new TicketController(ticketService);
```

Note: Normally, these instances are injected by Spring Boot automatically, but here, we are doing it manually.

The next step is to create repositories for gates, vehicles, parking lots, and tickets. However, we won't expose controllers at the moment. Instead, we'll focus on creating getters and setters for these repositories.

Go to GateRepository and create getters and setters for the map:----------

```
private Map<Long, Gate> gates = new HashMap<>();
public Map<Long, Gate> getGates() {
  return gates;
}

public void setGates(Map<Long, Gate> gates) {
  this.gates = gates;
}
```

In TicketRepository:----------

```
private Map<String, Ticket> ticketMap = new HashMap<>();

public Map<String, Ticket> getTicketMap() {
  return ticketMap;
}

public void setTicketMap(Map<String, Ticket> ticketMap) {
  this.ticketMap = ticketMap;
}
```

For VehicleRepository: —----------

```
private Map<String, Vehicle> vehicleMap = new HashMap<>();

public Map<String, Vehicle> getVehicleMap() {
  return vehicleMap;
}

public void setVehicleMap(Map<String, Vehicle> vehicleMap) {
  this.vehicleMap = vehicleMap;
}
```

ParkingLotRepository:----------

```
private Map<Long, ParkingLot> parkingLotMap = new HashMap<>();

public Map<Long, ParkingLot> getParkingLotMap() {
  return parkingLotMap;
}

public void setParkingLotMap(Map<Long, ParkingLot> parkingLotMap) {
  this.parkingLotMap = parkingLotMap;
}
```

These **getter and setter** methods allow external access to the internal maps in each repository, providing a way to retrieve and update the stored data.

The next step is to go back to the **App** class. The first thing we need to create is a parking lot, and we'll start by creating it at the top. To do this, we go into the **ParkingLot** class and link the id, **floors**, **gates**, and **parkinglotstatus** using a constructor that we generate.

```
private Long id;
private List<Floor> floors;
private List<Gate> gates;
private ParkingLotStatus parkingLotStatus;

public ParkingLot(Long id, List<Floor> floors, List<Gate> gates, ParkingLotStatus
            parkingLotStatus) {
  this.id = id;
  this.floors = floors;
  this.gates = gates;
  this.parkingLotStatus = parkingLotStatus;
}
```

Additionally, create a getter and setter for id.

The next step is to pass all this information, so we switch to the Gate class and link the number, gate type, gate status using a constructor that we generate. We'll keep the operator optional and not add it to the constructor.

```
private Long number;
private GateType gateType;
private Operator operator;
private GateStatus gateStatus;

public Gate(Long number, GateType gateType, GateStatus gateStatus) {
  this.number = number;
  this.gateType = gateType;
  this.gateStatus = gateStatus;
}
```

Now, go back to the **App** class and create a gate object.

```
Gate gate = new Gate(1L, GateType.ENTRY, GateStatus.WORKING);
```

Now, we need a **floor** object, and for this, we need the **spot** object. Create a parking spot object, and to do this, generate a constructor in **ParkingSpot**, linking the number, vehicle type, status, and floor.

```
private int number;
private VehicleType vehicleType;
private ParkingSpotStatus status;
private Floor floor;
```

```
public ParkingSpot(int number, VehicleType vehicleType, ParkingSpotStatus status, Floor
                   floor) {
  this.number = number;
  this.vehicleType = vehicleType;
  this.status = status;
  this.floor = floor;
}
```

Go back to the **App** class and create a parking lot object. To create this, we need a Floor, so switch to the Floor class and create a constructor, linking floor number. Then, come back to the App and pass the floor number to create the object just before the parking lot.

```
Floor floor1 = new Floor(1);
Now, we need to create multiple parking spots for this floor, and we'll achieve this
using a loop.

      for(int i=1; i<=10; i++) {
      floor1.getParkingSpots().add(
      new ParkingSpot(i, VehicleType.FOUR_WHEELER, ParkingSpotStatus.FILLED, floor1));
      }
```

The next step is to create floors and gates after the loop.

```
ArrayList<Floor> floors = new ArrayList<>();
floors.add(floor1);
ArrayList<Gate> gates = new ArrayList<>();
gates.add(gate);
```

Now, finally, create a parking lot by passing all the required objects and fields.

```
ParkingLot parkingLot = new ParkingLot(1l, floors, gates, ParkingLotStatus.ACTIVE);
```

After creating the parking lot, we have to save it in the database. Since we are not using a database, we will save it in the **parkingLotRepository** from the App using the put method.

```
parkingLotRepository.getParkingLotMap().put(1L, parkingLot);
```

Similarly, the newly created gate should be present in the gate repository. So, we do the same as we did for the parking lot using the put method.

```
GateRepository gateRepository = new GateRepository();
gateRepository.getGates().put(1L, gate);
```

Note: As we are passing the gate number as long, go to the repository and change the required type.

The next step is to issue a ticket. For this, we go into the IssueTicketRequest and link vehicle number, gate ID, parking lot ID, and parking place allotment strategy by generating a constructor for mandatory fields.

```
public IssueTicketRequest(String vehicleNumber, Long gateId, Long parkingLotId,
             ParkingPlaceAllotmentStrategy parkingPlaceAllotmentStrategy) {
  this.vehicleNumber = vehicleNumber;
  this.gateId = gateId;
  this.parkingLotId = parkingLotId;
  this.parkingPlaceAllotmentStrategy = parkingPlaceAllotmentStrategy;
}
```

Create one more constructor for all fields.

```
public IssueTicketRequest(VehicleType vehicleType, String vehicleNumber, String owner,
                 Long gateId, Long parkingLotId, ParkingPlaceAllotmentStrategy
                 parkingPlaceAllotmentStrategy) {
  this.vehicleType = vehicleType;
  this.vehicleNumber = vehicleNumber;
  this.owner = owner;
  this.gateId = gateId;
  this.parkingLotId = parkingLotId;
  this.parkingPlaceAllotmentStrategy = parkingPlaceAllotmentStrategy;
}
```

Now, come back to the **App** and create an instance for issuing a ticket.

```
IssueTicketRequest issueTicketRequest = new IssueTicketRequest(
      VehicleType.FOUR_WHEELER, "KA-02",
      "Keerthi", 1l, 1l, new SimpleParkingSpotAllotmentStrategy());
IssueTicketResponse ticketResponse = ticketController.issueTicket(issueTicketRequest);
System.out.println(ticketResponse);
```

App Summary:

First, we created a ticket controller. Then, we developed a ticket service. To set up this service, we needed repositories for gates, vehicles, and parking lots. In order to create these repositories, we also generated data for gates, parking spots using a loop, and floors, adding them to the repositories. This way, when we generate a ticket, we can access the necessary details.

However, there is an issue: we haven't implemented a strategy. So, we implemented the **SimpleParkingSpotAllotmentStrategy** by checking the availability of a parking spot.

```
public class SimpleParkingSpotAllotmentStrategy implements ParkingPlaceAllotmentStrategy  {
```

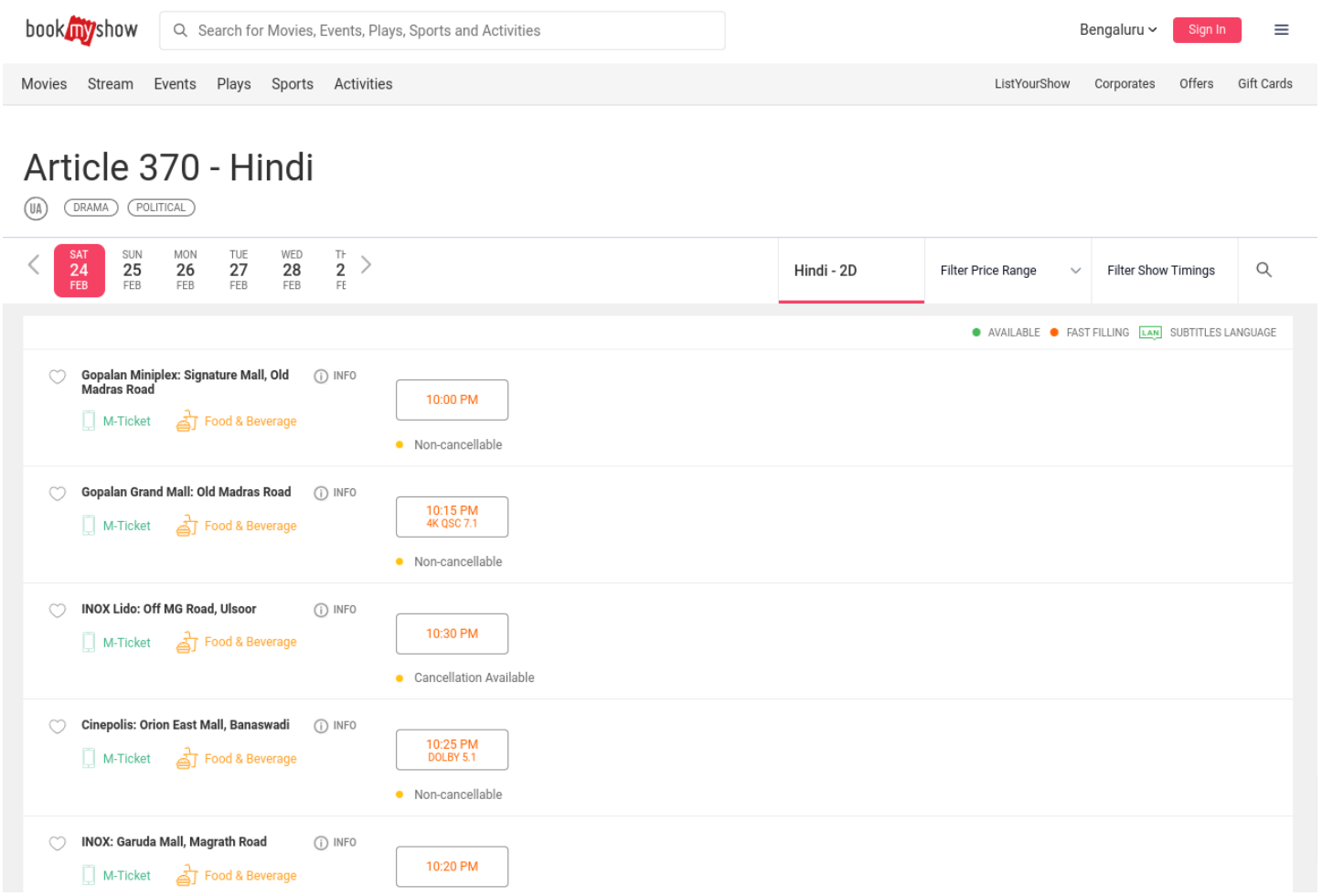
```
    @Override
    public ParkingSpot getParkingSpot(VehicleType vehicleType, ParkingLot parkingLot)
            throws ParkingLotFullException {

        /*
          1. Get all the floors of the parking lot
             For each floor, check the slots with vehicle type and status being free
             Return it.
         */

        for(Floor floor: parkingLot.getFloors()){
            for(ParkingSpot parkingSpot: floor.getParkingSpots()){
                if(parkingSpot.getStatus().equals(ParkingSpotStatus.AVAILABLE)){
                    if(parkingSpot.getVehicleType().equals(vehicleType)){
                        return parkingSpot;
                    }
                }
            }
        }

        // throw an exception saying the parking lot is full.
        throw new ParkingLotFullException();
    }
}
```

Now, run the app and get a ticket response. If there is any null pointer, simply go into the file and assign an empty list using the constructor.
https://github.com/kkumarsg/parkingLotMorning

Backend LLD-3: Machine Coding - 9: Design BookMyShow - Feb 22

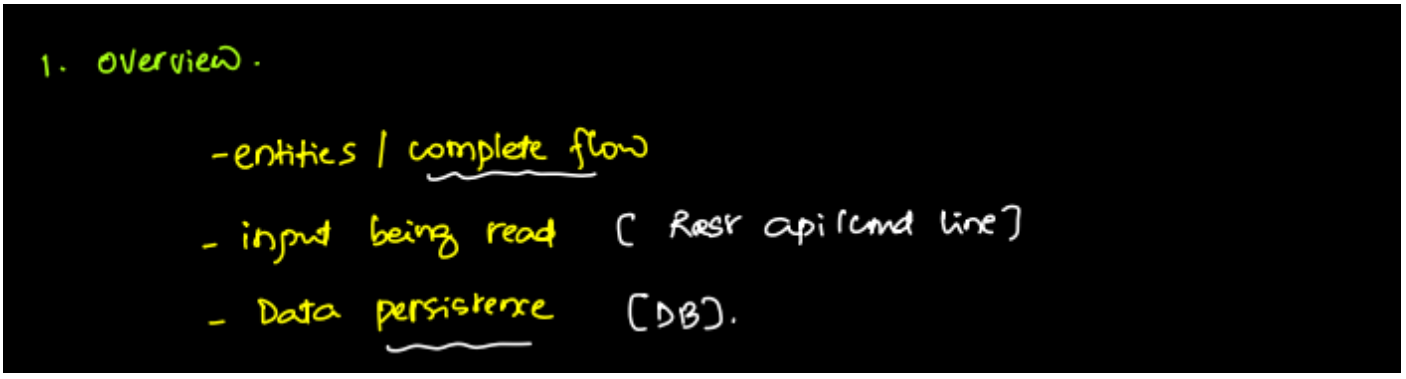
- **Visit bookmyshow and try to capture the requirements and Exploring book my show**
- **Questions to ask in overview**
- **Requirement gathering**
- **Who owns the data of seats being booked.**
    - **We're assuming BMS owns the data, although in reality BMS is an aggregator**
- **Messing up with BMS and PVR cinemas**
- **Class diagram**
- **Seat class**
- **Show and ShowSeat**
- **Where to keep the price ?**
- **User, Ticket/Booking/, Payment**

Visit bookmyshow and try to capture the requirements and Exploring book my show

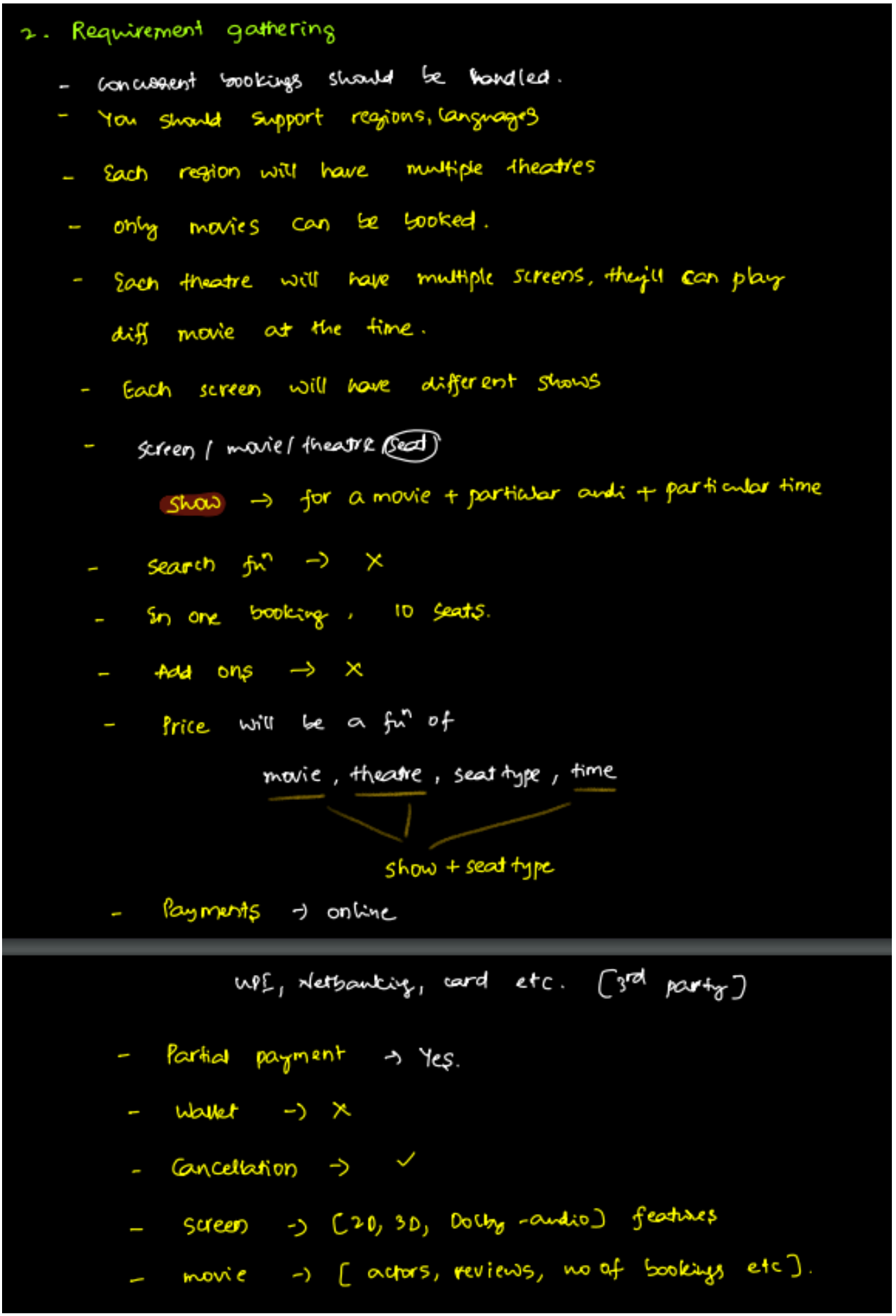


When we explore BookMyShow, there are many movies and cities to choose from. If we click on a movie, we can book tickets and see various shows in different theaters. Each theater screen displays different shows, along with details like the screen type, movie name, and showtime. There's also a search option for easy browsing. We can check the prices, and there's a limit of booking up to 10 tickets. After that, we can view the theater layout, showing seats with different prices in each section. Once we pick our seats, we go to the payment page where we can add extras. It asks for our personal details before making the payment through various options. We've looked at the whole process and now aim to implement a feature to generate tickets based on what we observed.

Questions to ask in overview

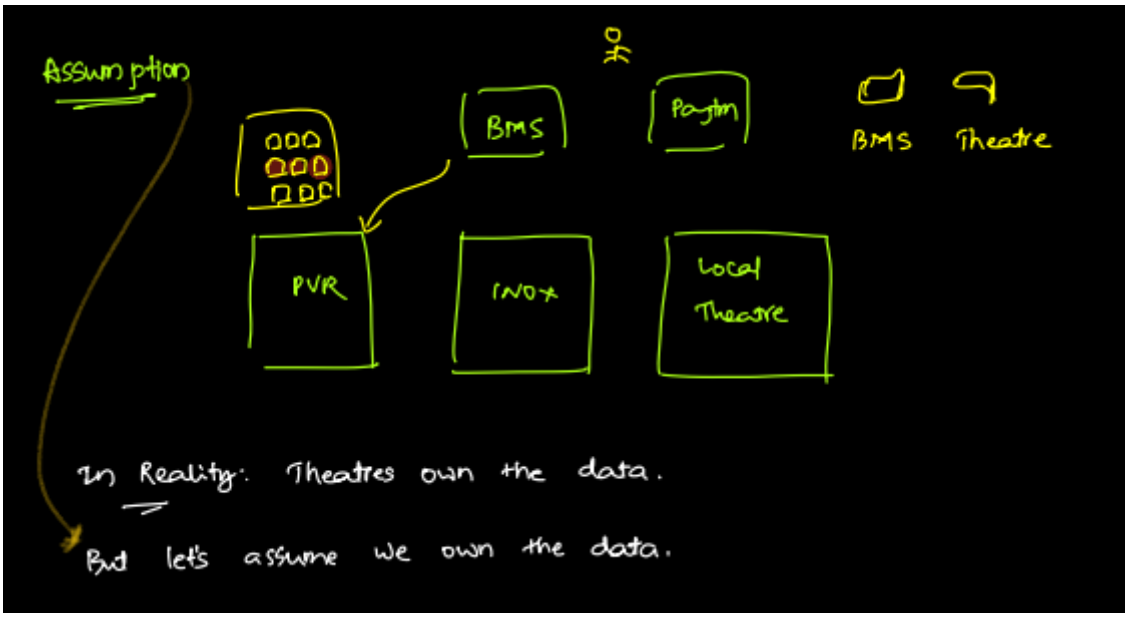


Requirement gathering

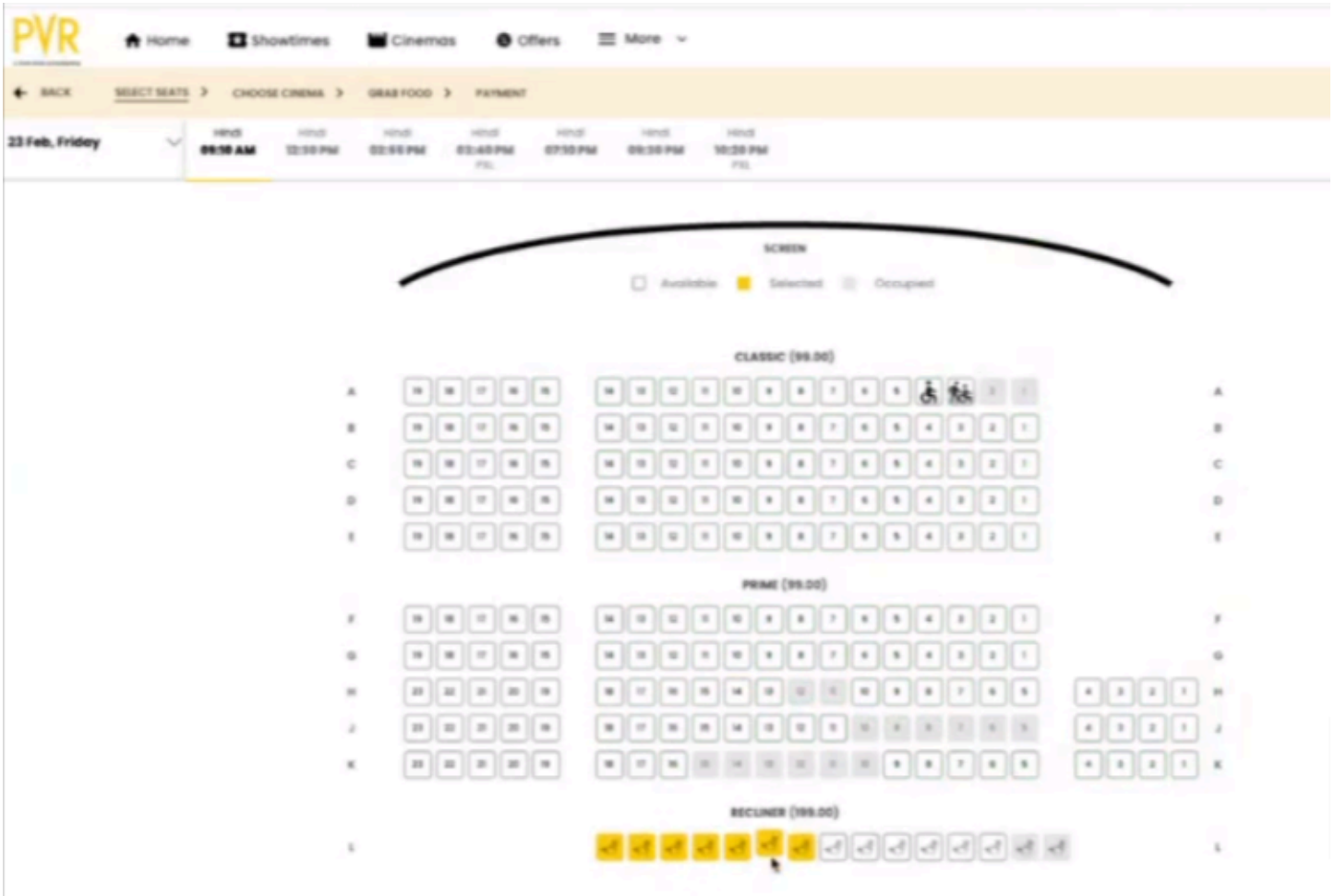


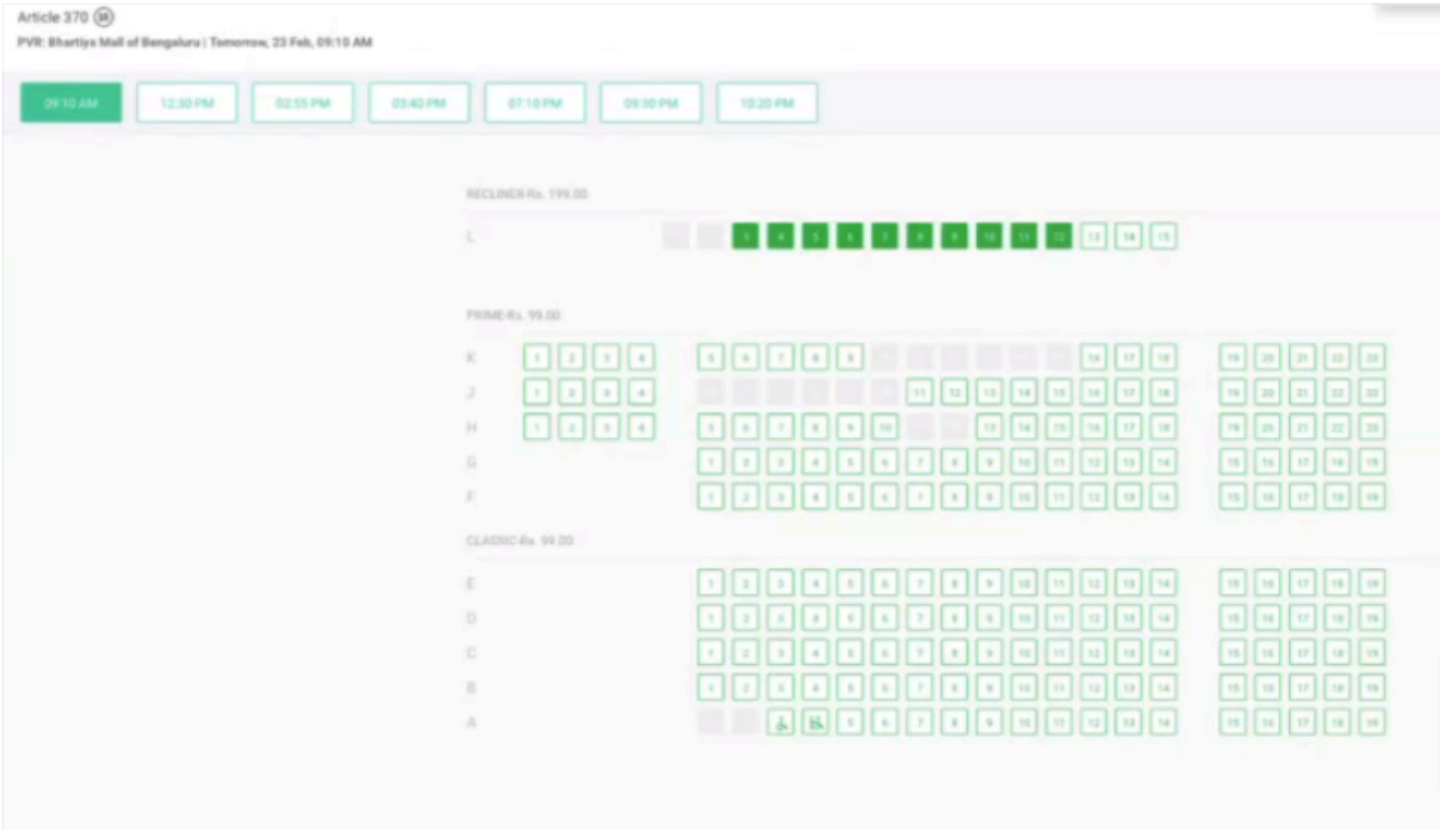
Who owns the data of seats being booked ->

When we book a ticket, the theater owns all the data, and BookMyShow gets that information and booking details using a special tool called an API provided by the theater. Since we don't have that tool, we'll assume that all the information, like movies, actors, seat layout, and seat selection, belongs to BookMyShow.



Messing up with BMS and PVR cinemas

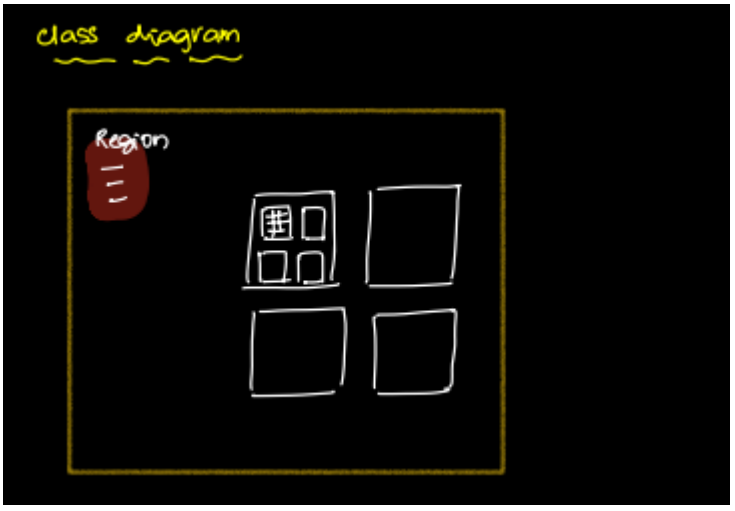




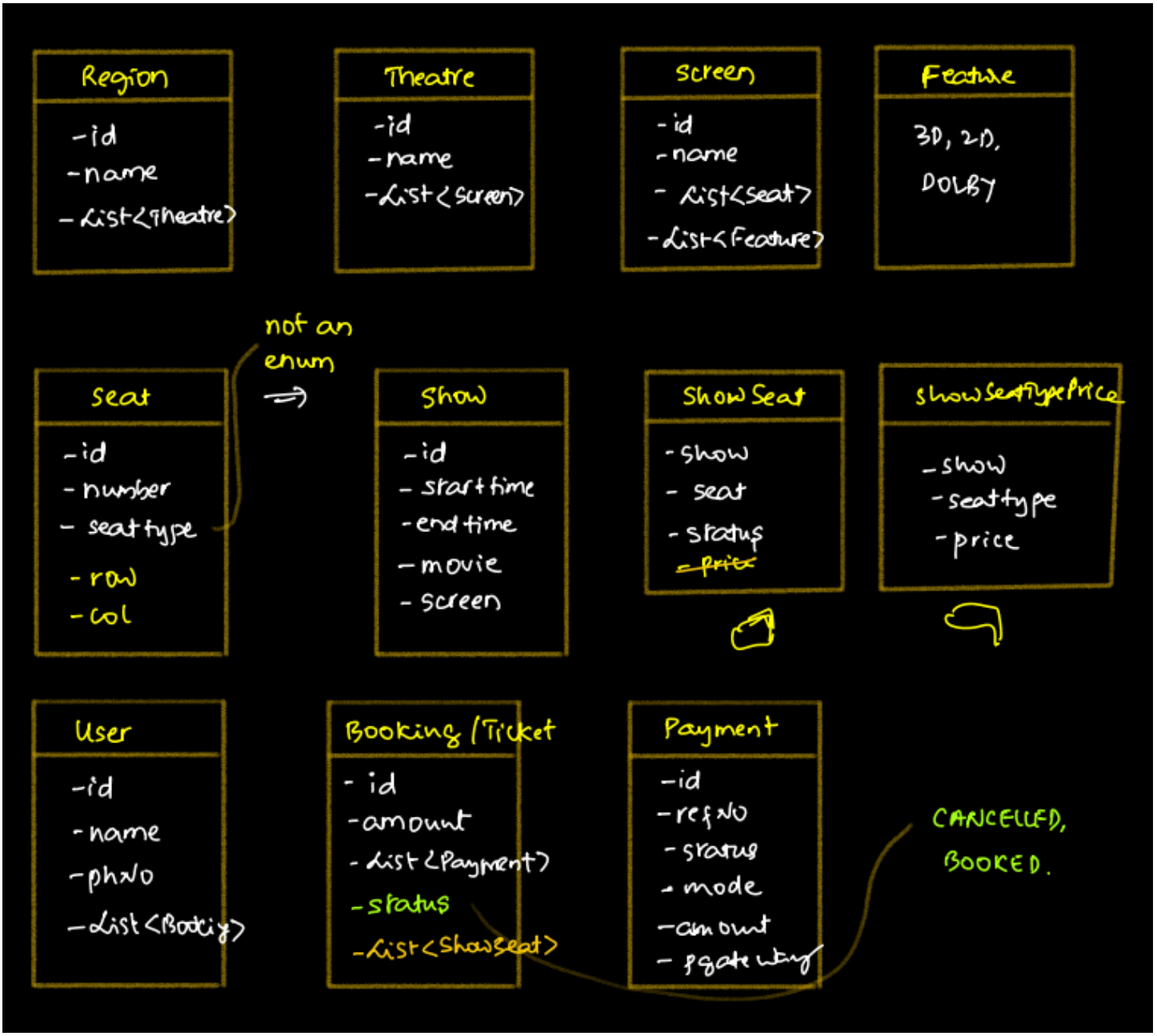
In both screens, we can pick the same seat, but when it's time to pay, only one person is allowed, and the other person gets an error. This could be because they're using a system that manages multiple actions at the same time, like a synchronized lock.

Class diagram

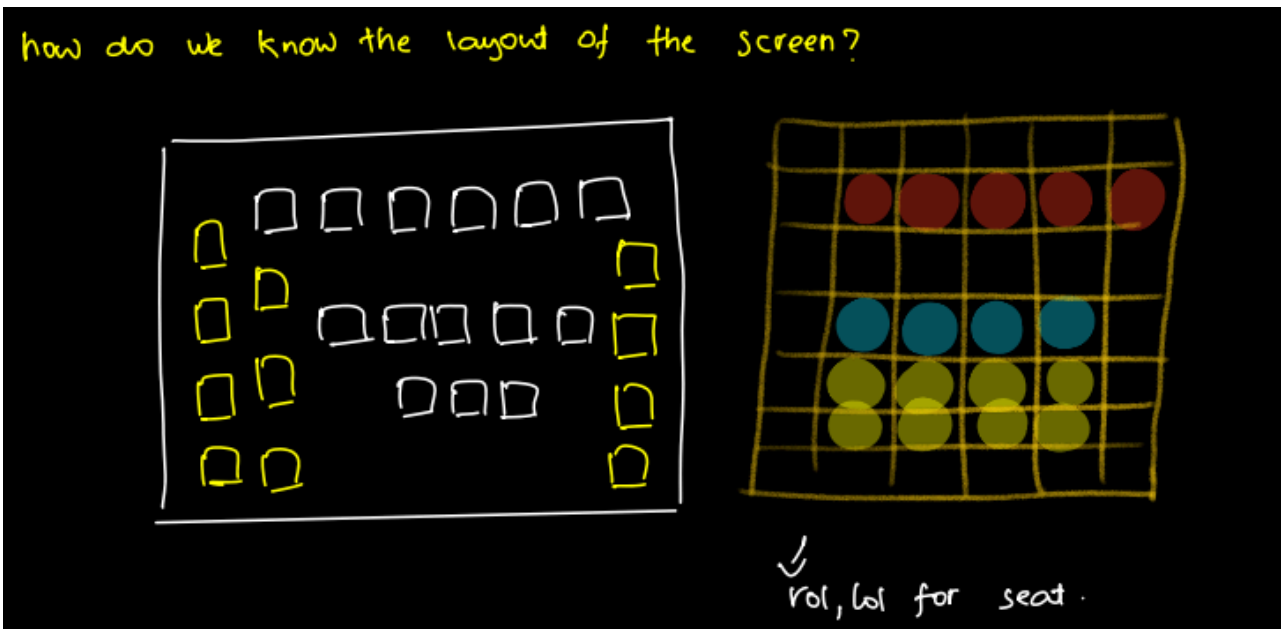
When making a class diagram, we have two ways to go about it. If there are specific requirements given, we look for entities (things with properties and actions, like living or non-living stuff, called entities). If there are no requirements provided, we use visualization to create the diagram since we don't have specific instructions to follow.



In the design of BookMyShow above, we see different regions, and each region has multiple theaters. Inside each theater, there are multiple screens and shows. Now, the question is, do we need to make a copy of BookMyShow, similar to a parking lot? Since there's only one app here, it's not necessary to create a separate instance.


Region
-id
-name
-List<Theatre>
Theatre
-id
-name
-List<Screen>
Screen
-id
-name
- List<Seat>
-List<Feature>
Feature
3D, 2D,
DOLBY
not an enum
Seat
-id
- number
- seat type
- row
- col
Show
-id
- start time
-end time
- movie
- screen
Show Seat
- show
- seat
- status
- price
ShowSeatTypePrice
- show
- seat type
- price
User
-id
- name
-phNo
-List<Booking>
Booking / Ticket
- id
-amount
- List<Payment>
-status
-List<ShowSeat>
Payment
-id
-refNo
- status
- mode
-amount
- gateway
CANCELLED,
BOOKED.

Seat class ->

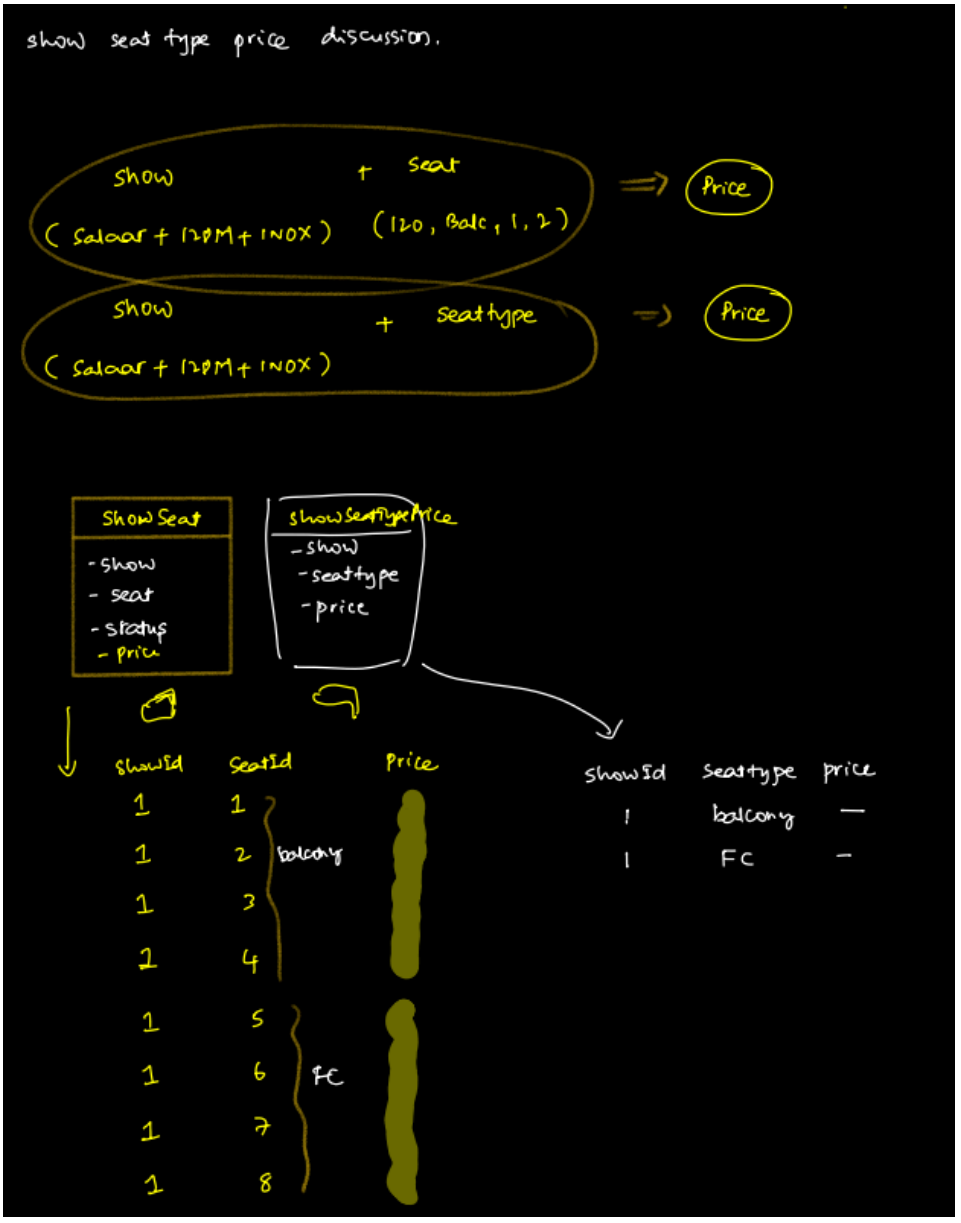

how do we know the layout of the screen?
rol, col for seat.

We won't make "seat" an enum because each theater has its own way of naming seats, like "walkani," "maharaja," or "queen," with different properties.

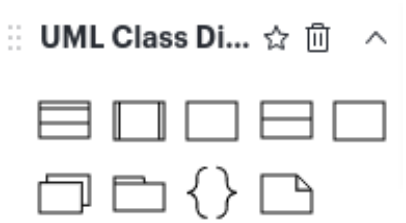
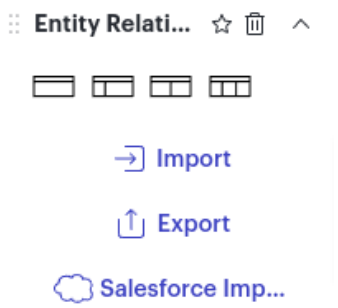
Also, we won't tie the status directly to the seat because a seat can have various statuses in different shows. For example, a seat can be booked in one show but not in another. We'll include two more essential properties, row and column, to help locate the seat and arrange the layout as needed.

Show and ShowSeat -> We shouldn't link the status of a seat directly to a show. The challenge is figuring out how and where to manage this status because one show can involve many seats, and one seat can be part of many shows. To handle this, we'll introduce another class called ShowSeat, which stores information about the show, seat, and status.

Where to keep the price ? ->
Now, the question arises: should we include the price in ShowSeat? Since the price depends on various factors like seat, show, and timings, it's better to create another class specifically for pricing to ensure better organization. Instead of using an enum for seat type, we'll create a separate class and include it in ShowSeatTypePrice, where we store details about the show, seat type, and price.



I start creating a class diagram by first jotting down all the requirements in a notepad. Once I've listed all the entities, I begin designing the class diagram using https://lucid.app/, we also can sue https://draw.io/. To get started, open the website, create an account, and refer to the "Entities" section for creating a class diagram. For schema design, gather elements from the entities in the ER (Entity-Relationship) section.

For Class Diagram use	for Schema Duagram use
	

City

---

name, List<Theatre>

Theatre

---

name, List<Screen>

Screen

---

name, List<Seat>, List<ScreenType>

ScreenType (enum)

2D, 3D, DOLBY_IMAX, DOLBY_VISION

Seat

---

seatNum, seatType, row, col

SeatTypeName (enum)

PLATINUM, GOLD, SILVER

Show

---

screen, movie, startTime, endTime

ShowSeat

---

show, seat, showSeatStatus

ShowSeatStatus

BOOKED, AVAILABLE, RESERVED, NOT_AVAILABLE

ShowSeatTypePrice

---

day, SeatType, price

Day (enum)

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

SeatType

---

seatTypeName, size

Movie

---

title, genre, duration

Booking

---

show, List <ShowSeat>, status, amount

BookingStatus

PENDING, CONFIRMED, CANCELED

Payment

---

amount, transactionId, Booking, payment_status, paymentMode

PaymentMode (enum)

Razorpay

PaymentStatus (enum)

PENDING, COMPLETED, FAILED

User

---

name, contact, email

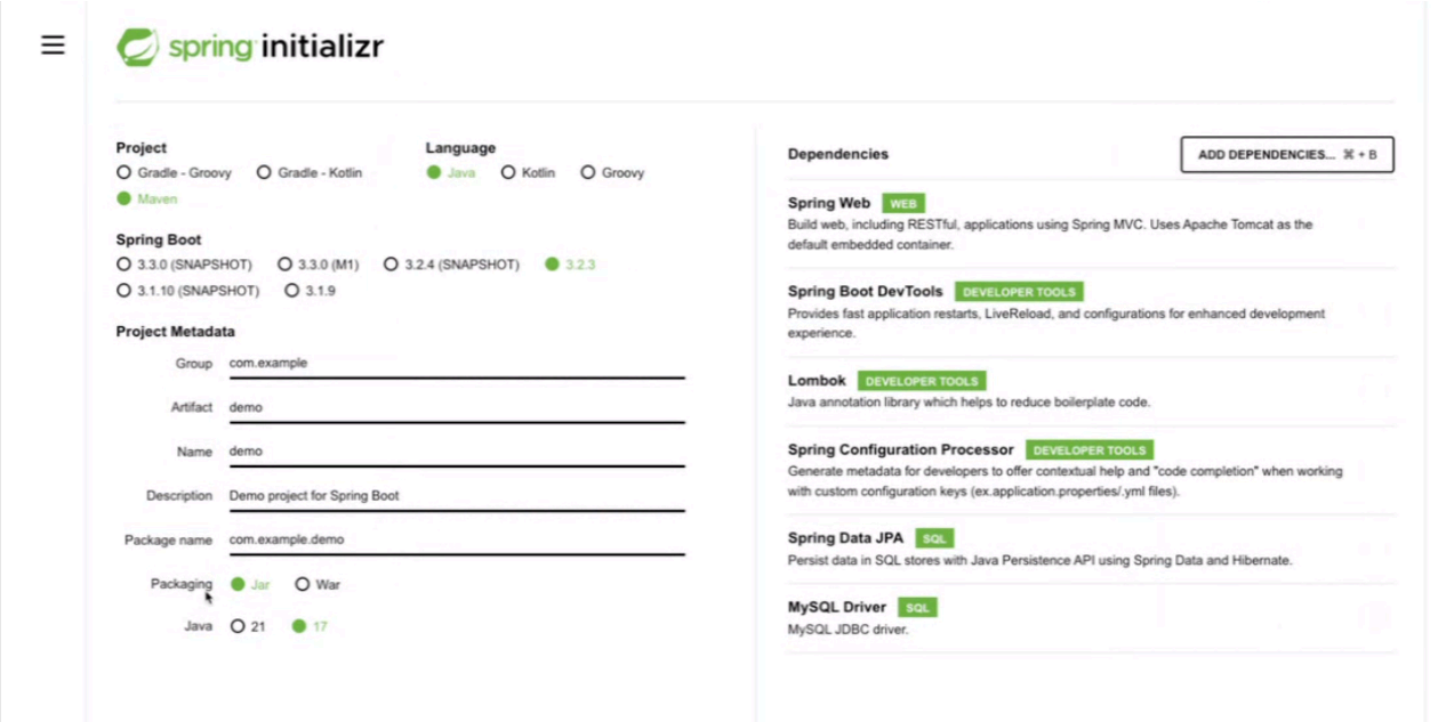
Backend LLD-3: Machine Coding - 10: Code BookMyShow 1 - Feb 24

- **Creating a spring boot project**
- **Importing the project we created from spring initializer**
- **The application fails to start because we haven't given the database configurations**
- **Creating baseModel**
- **Lombok to generate getters and setters**
- **Primary key for the table**
- **Creating all the Models**
- **@Entity for all the models**
- **How to map enums to tables ? @Enumerated**

Creating a spring boot project ->

1. **Go to https://start.spring.io/**: Visit the Spring Initializer website at https://start.spring.io/ to create a new Spring Boot project.
2. **Configure your project settings**: On the Spring Initializer website, you'll see various configuration options. Fill in details like project type (Maven or Gradle), language (Java or Kotlin), Spring Boot version, project metadata (like group name, artifact name, etc.), and packaging format (JAR or WAR).
3. **Click the "Add Dependency" button and add the following dependencies:**
    - **Spring Web:** This dependency includes features for building web applications with Spring MVC. It facilitates the development of RESTful services and web applications.
    - **Spring Boot DevTools:** DevTools provide a set of development-time tools that improve the development experience. It includes features like automatic application restart, among others.
    - **Lombok:** Lombok is a library that helps reduce boilerplate code in Java classes. It provides annotations to automatically generate getter, setter, toString, and other common methods.
    - **Spring Configuration Processor:** The Configuration Processor helps with generating metadata files to make configuration properties more user-friendly.
    - **Spring Data JPA:** Spring Data JPA simplifies data access using the Java Persistence API (JPA). It provides a set of abstractions and implementations for working with relational databases.
    - **MySQL Driver:** This dependency includes the MySQL JDBC driver, which is required to connect a Spring Boot application to a MySQL database.
4. **After configuring and adding dependencies, click the "Generate Project" button:** Once you've set up your project details and added the necessary dependencies, click the "Generate Project" button. This action downloads a zip file containing the initial structure and configuration files for your Spring Boot project.

In summary, these steps help you set up a Spring Boot project with specific configurations and dependencies to ease the development of web applications with features like web services, automatic restart during development, reduced boilerplate code, data access with JPA, and MySQL database connectivity.



Importing the project we created from spring initializer

After downloading, unzip the file, open IntelliJ, locate the pom.xml file, and open it to initiate the project in IntelliJ instead of through the directory. When we open the pom.xml file, it begins installing dependencies. The pom.xml file is crucial as it is used to add dependencies, similar to the package.json file in JavaScript.

The POM (Project Object Model) file in Maven is a fundamental configuration file that contains essential information about a Maven project. Here's an explanation of its key components:

1. **Project Information:**
    - **Group Id, Artifact Id, Version**: Identifies the project uniquely. The combination of these three elements forms a unique identifier for the project.
    - **Packaging**: Specifies the packaging format for the project (e.g., JAR, WAR). It defines how the project should be packaged and delivered.
2. **Dependencies**:
    - **Dependencies Section**: Lists all the external libraries and dependencies required for the project. Maven uses this information to automatically download and manage these dependencies during the build process.
3. **Build Configuration**:
    - **Build Section**: Configures the build process, including plugins and goals. Maven uses plugins to execute various tasks during the build, such as compiling code, running tests, and packaging the project.
4. **Repositories**:
    - **Repositories Section**: Specifies the remote repositories where Maven can find and download dependencies. It includes default repositories like Maven Central and allows you to define custom repositories.

5. **Profiles**:
    - **Profiles Section**: Defines different sets of configurations that can be activated based on certain conditions. Profiles are useful for managing variations in the build process, such as development, testing, or production environments.
6. **Properties**:
    - **Properties Section**: Allows the definition of variables that can be used throughout the POM file. It helps in making the configuration more dynamic and reusable.

The POM file acts as the project's blueprint, providing Maven with instructions on how to build, package, and manage dependencies for the project. It simplifies project management by standardizing project structure, dependency management, and build processes. Additionally, it promotes consistency across projects and facilitates collaboration among developers.

The application fails to start because we haven't given the database configurations
When we open a project, it typically displays a class named GivenNameApplication with a main method. This class usually contains a single line:
SpringApplication.run(GivenNameApplication.class, args);.

However, when we run this application, an error message might appear, stating "Application failed to start." The reason behind this error is the failure to determine a suitable driver class. This indicates that the application is attempting to connect to a data source or database, but it's unable to identify an appropriate driver class. Notably, there's no visible URL for the connection. We can configure it later.

Creating baseModel -> creating a BaseModel involves defining a class that serves as a base or parent for other domain models. This class typically includes common attributes and methods that are applicable to various entities within the application.

```
import javax.persistence.*;
import java.util.Date;

@Data
@MappedSuperclass
public abstract class BaseModel {

  // ORM - Hibernate
  // Object Relation Mapping
  // Class : Table, Mapping -> ORM

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Temporal(TemporalType.TIMESTAMP)
  @Column(name = "created_at", nullable = false, updatable = false)
  private Date createdAt;
```

```
@Temporal(TemporalType.TIMESTAMP)
  @Column(name = "updated_at")
  private Date updatedAt;

  // Constructors

  public BaseModel() {
      this.createdAt = new Date();
  }

  // Other common methods can be added as needed
}
```

In this example:

- The **@MappedSuperclass** annotation is used to indicate that this class will be mapped to the database but is not an entity itself.
- The **@Id** annotations define the primary key **@GeneratedValue** provide the strategy to auto generate id, and **@Temporal** annotations specify the temporal type for date fields.
- The BaseModel includes common fields like id, createdAt, and updatedAt.
- It automatically sets the createdAt field when an instance is created.
- The **@Data** annotation from **Lombok** includes **@Getter** and **@Setter**, automatically generating getters and setters for all fields in the class.
- The **@Data** annotation also includes **@ToString, @EqualsAndHashCode**, and **@NoArgsConstructor** annotations, providing additional functionality such as a toString method, an equals method, and a no-args constructor.
- Other models in the project can extend BaseModel to inherit these common attributes and behaviors.

For instance, a User class might extend BaseModel:

```
@Entity
@Table(name = "users")
public class User extends BaseModel {

  // User-specific attributes, constructors, getters, setters, etc.
}
```

This approach helps in maintaining a consistent structure and behavior across different entities in the Spring project. Additionally, it leverages JPA annotations for database mapping.

Hibernate -> Spring Boot, Hibernate is often used as the default ORM (Object-Relational Mapping) framework for database interactions. Spring Boot simplifies the integration of Hibernate by providing auto-configuration and defaults that work seamlessly with various databases.

Here's how Hibernate is commonly used with Spring Boot:

Spring Data JPA: Spring Boot works well with Spring Data JPA, which is a part of the larger Spring Data project. Spring Data JPA is a data access abstraction that provides a consistent and familiar way to interact with different types of databases using the Java Persistence API (JPA).

Entity Classes: In a Spring Boot application, you define entity classes to represent your data model. These classes are typically annotated with JPA annotations (which Hibernate recognizes) to define the mapping between Java objects and database tables.

Repository Interfaces: Spring Data JPA provides repository interfaces that allow you to perform standard database operations (CRUD) without writing explicit SQL queries. You simply define repository interfaces and Spring Data JPA generates the necessary queries.

Auto-Configuration: Spring Boot's auto-configuration feature automatically configures Hibernate and the DataSource based on the application's dependencies and properties. This reduces the need for extensive manual configuration.

Step by Step Guide for Book My Show ->
Source: https://github.com/kkumarsg/bookmyshow1

Next Step: Setup Spring Boot Project: Create a new Spring Boot project using your preferred IDE or https://start.spring.io/.

Next Step: BaseModel.java in Models: Inside the Models package, create a BaseModel.java
Create a BaseModel Class:Make BaseModel abstract, if needed.
Add the following code to define the primary key and auto-generation strategy for the id field:

```
import lombok.Getter;
import lombok.Setter;
import javax.persistence.*;
import java.util.Date;

@MappedSuperclass
@Getter
@Setter
public abstract class BaseModel {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;

  private Date createdAt;
  private Date modifiedAt;
}
```

Create a new class **Region** extending BaseModel:

- Inside the Models package, create a Region.java file.
- Extend Region from BaseModel.
- Add the @Getter and @Setter annotations to automatically generate getter and setter methods.

```
import lombok.Getter;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "regions")
@Getter
@Setter
public class Region extends BaseModel {
  private String name;
}
```

Create a new class **Theatre** extending BaseModel:

```
@Getter
@Setter
@Entity
public class Theatre extends BaseModel{

   private String name;
   private Region region;
   private List<Screen> screenList;
}
```

Same create all the remaining models

Screen
private List<Seat> seatList;
private List<Feature> featureList;

Seat
private int seatNumber;
private SeatType seatType;
private int rowNum;
private int colNum;
Feature (enum)
DOLBY, TWO_D, THREE_D

SeatType
private String name;

Show -> The "Show" could be causing an error because of reserved keyword, so we're considering changing its name.

```
@Getter
@Setter
@Entity(name="show_bms")
public class Show extends BaseModel{
  private Date startTime;
  private Date endTime;
  @ManyToOne
  private Movie movie;
  @ManyToOne
  private Screen screen;
}
```

Movie
private String name;
private String description;
private int releaseYear;

ShowSeat
private Show show;
private Seat seat;
private ShowSeatStatus showSeatStatus;

ShowSeatStatus (enum)
BOOKED, EMPTY, BROKEN

ShowSeatType
private Show show;
private SeatType seatType;
private int price;

User
private String name;
private String phoneNumber;
private List<Booking> bookingList;

Booking
private BookingStatus bookingStatus;
private int amount;
private List<Payment> paymentList;
private List<ShowSeat> showSeatList;

BookingStatus (enum)
BOOKED, WAITING, FAILED, CANCELLED

Payment
private String refNo;
private PaymentStaus paymentStaus;
private String gateWay;
private PaymentMode paymentMode;
private int amount;

PaymentStaus (enum)
DONE, WAITING, FAILED

PaymentMode **(enum)**
CARD, UPI, NET_BANKING

Next Step: To transform a all model into a table, include the `@Entity` annotation on top of every model. After adding this whenever we run app it will automatically created in linked database.

NEXT Step: At every place where we declared ENUMS just above the declaration like BookingStatus in Boking class, we will specify the annotation **@Enumerated(EnumType.ORDINAL)**. This annotation allows us to choose between two options: ORDINAL or STRING. When using ORDINAL, the enum values are assigned numeric indices (0, 1, 2, etc.). Alternatively, if STRING is chosen, the enum behaves like strings. By using ORDINAL It will save id of ENUM instead String.

```
@Enumerated(EnumType.ORDINAL)
private BookingStatus bookingStatus;
```

If there is a list of enums, we need to specify **@Enumerated(EnumType.ORDINAL)** first, followed by **@ElementCollection**. For instance, in the code snippet, it would look like this: **private List<Feature> featureList**; in Screen class. where Feature is an enum.

```
import java.util.List;

@Enumerated(EnumType.ORDINAL)
@ElementCollection
private List<Feature> featureList;
```

Next step: In the Booking class, where we've defined List<Payment>, we need to inform the Object-Relational Mapping (ORM) system about how the Booking and Payment entities should be mapped. For every non-primitive type, we have to specify the cardinality or the relationship between them. like OneToMany, ManyToOne, ManyToMany

Booking

```
@OneToMany
private List<Payment> paymentList;
@OneToMany
private List<ShowSeat> showSeatList;
```

Screen

```
@OneToMany
private List<Seat> seatList;
```

Seat

```
@OneToOne
private SeatType seatType;
```

Sometimes, if we want to associate an item in a class with a mapped class, we can inform the Object-Relational Mapping (ORM) system by using the mappedBy attribute, specifying the name of the column. After adding this, the ORM system will refrain from creating duplicate columns. **but we will not implement it here**

```
public class Region extends BaseModel{

    private String name;
    @OneToMany(mappedBy = "region_id")
    private List<Theatre> theatreList;

}
```

Show

```
@ManyToOne
private Movie movie;
@ManyToOne
private Screen screen;
```

ShowSeat

```
@ManyToOne
private Show show;
@ManyToOne
private Seat seat;
```

ShowSeatType

```
@ManyToOne
private Show show;
@ManyToOne
private SeatType seatType;
```

Theatre

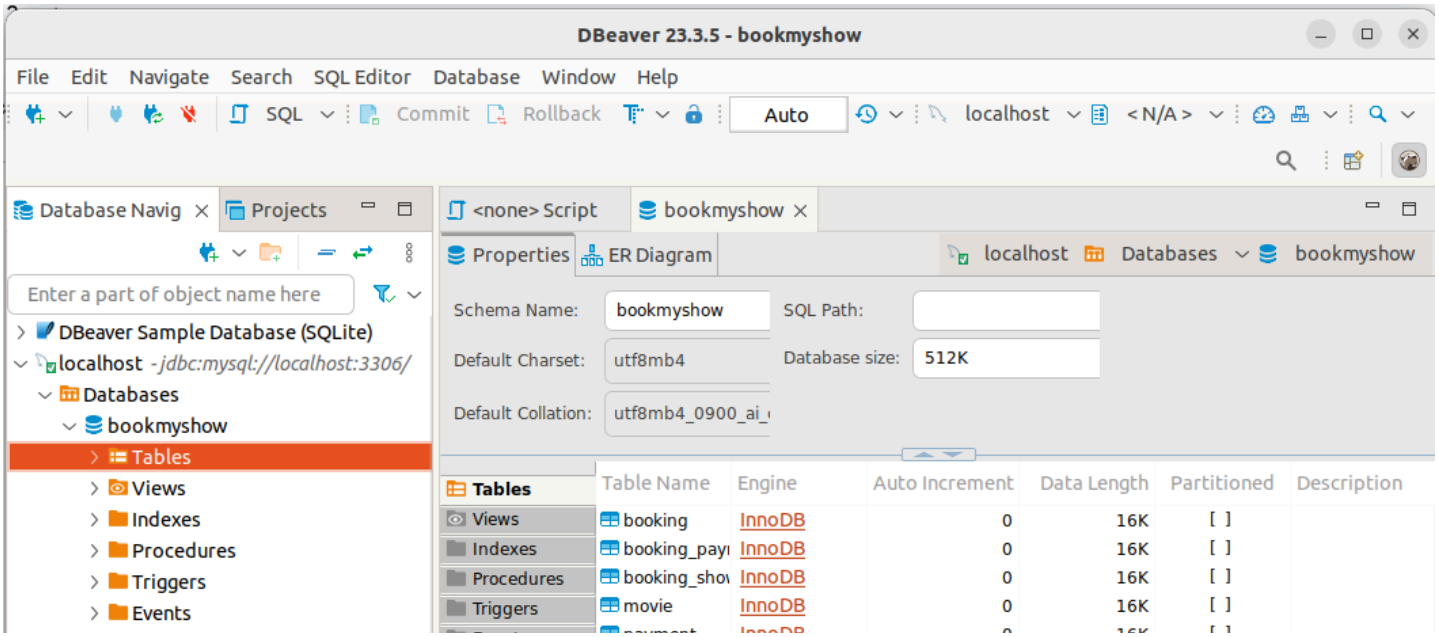
```
@ManyToOne
private Region region;
@OneToMany
private List<Screen> screenList;
```

User

Here we can observe @Entity(name = "user_bms"). We are specifying "BMS," and if we wish to change this table name because "user" could be a default SQL keyword, we can do so by providing a different name.

```
@Entity(name = "user_bms")
public class User extends BaseModel {
  private String name;
  private String phoneNumber;

  @OneToMany
  private List<Booking> bookingList;
}
```

Next step: **Establish a connection to the database.**
Begin by open MySQL Database Workbench or Dveaver. To establish the connection with springboot, you will need three essential pieces of information: URL, username, and password. Configure these parameters in the application.properties file. The file contains predefined names for easier setup. If database not present create new.

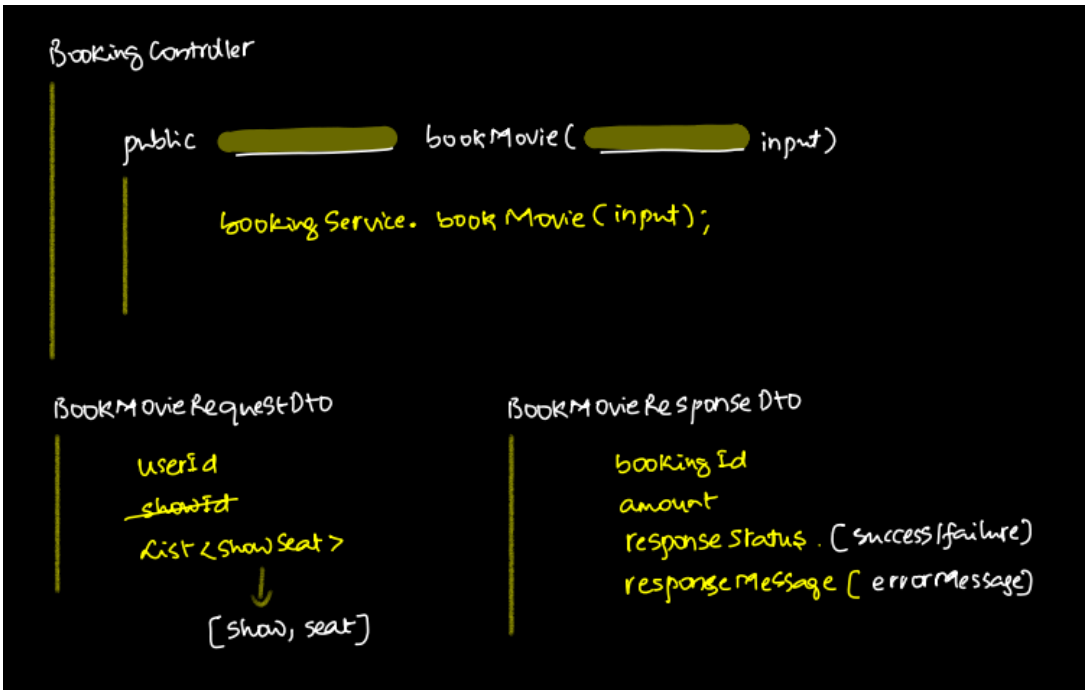



```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/bookmyshow1
spring.datasource.username=root
spring.datasource.password=
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=update
```

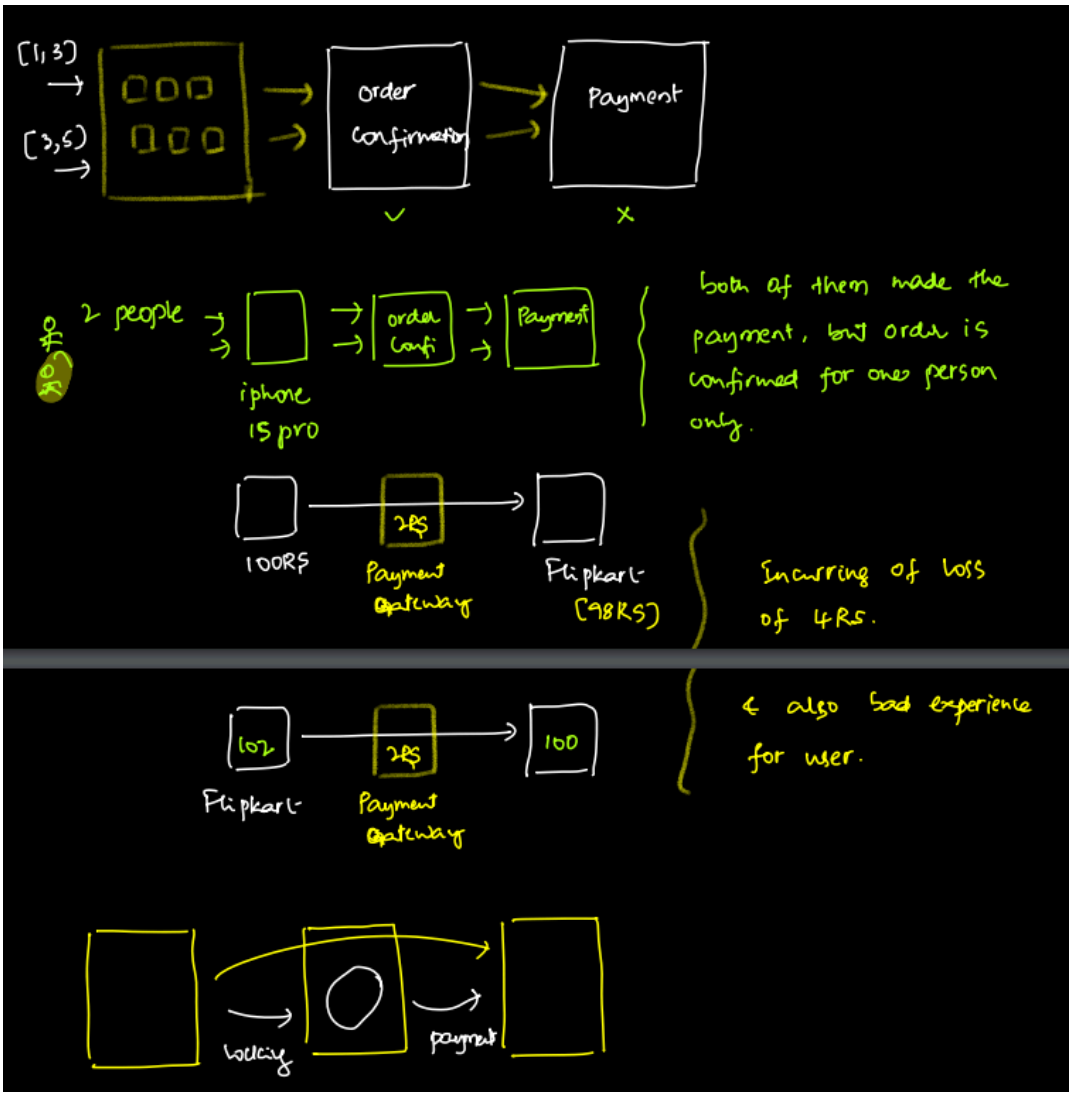
Backend LLD-3: Machine Coding - 11: Code BookMyShow 2 - Feb 27

- **Booking controller request and response**
- **What happens when you click on book a ticket**
- **Why a order confirmation page is always preferred**
- **DB transactions for locking**
- **Can long running transactions solve the issue ?**
- **Soft locking**
- **What if after taking the lock the payment never happens**
- **What needs to be done in service**
- **Repositories**
- **JPARepostory power demo**

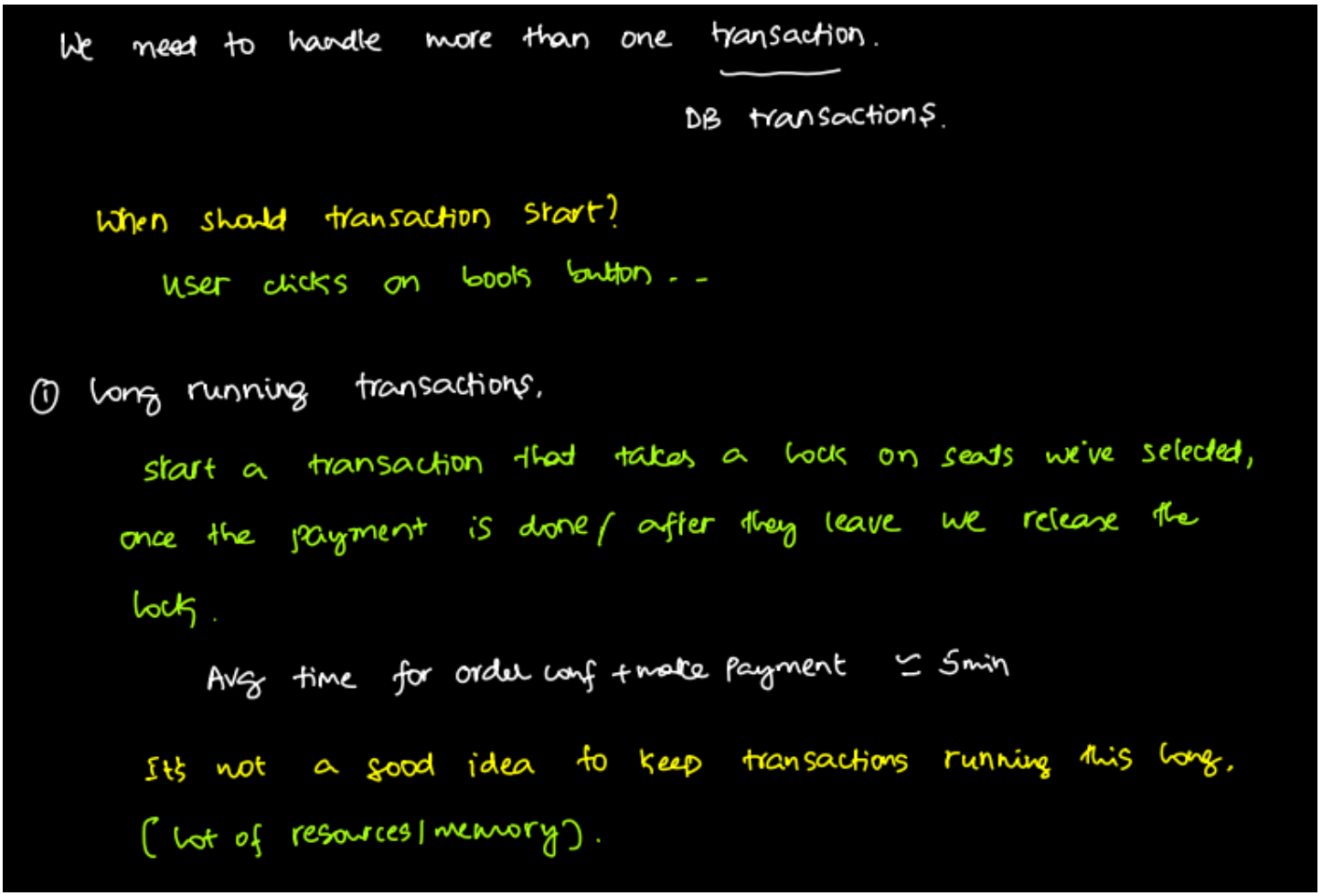
Booking controller request and response -> The flow begins with the controller, named BookingController, where we establish a method called bookMovie. This method takes a request DTO and returns a response DTO. To implement this, we will create a package named dto and include these two files as outlined in the screen below.



Why a order confirmation page is always preferred -> It's better to put a lock on the confirmation page instead of the payment page. If we use the payment page for the lock, multiple users might go to the payment page at the same time, causing multiple payments. In this situation, after only one successful payment, we'd have to cancel the others and refund the money, which would cost the company more and provide a not-so-good experience for users.

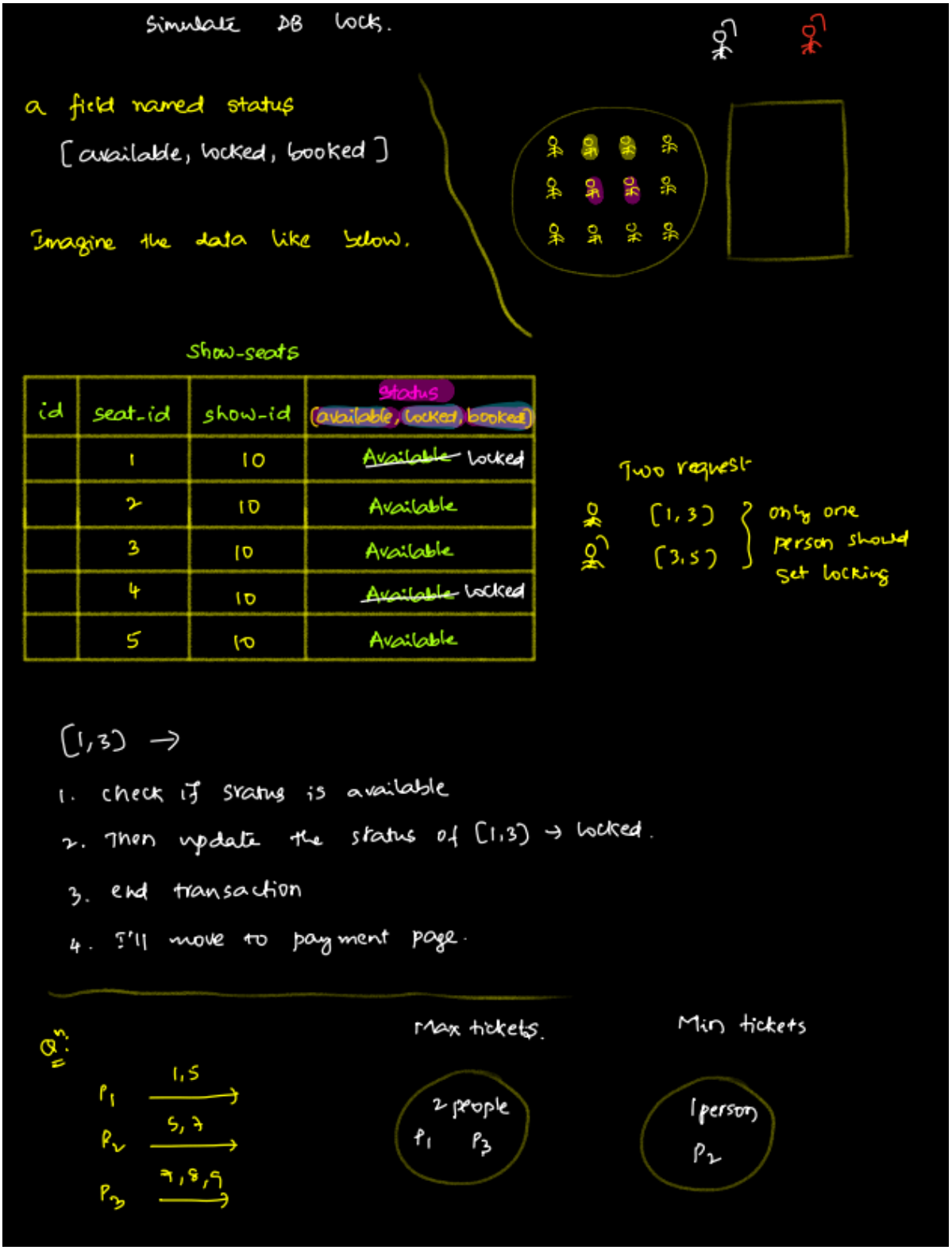


DB transactions for locking -> To manage multiple bookings, we should use a transaction. If any booking fails, we can roll back the entire process. To achieve this, we can implement a long-running transaction. For example, when a user selects a seat, we acquire a lock, and after completing the payment and confirming the booking, we release the lock. However, this method has a drawback as it increases the waiting time for users and makes it challenging to book seats. Moreover, it demands a significant amount of resources.



Soft locking -> As shown in the screen below, imagine two recruiters from different companies wanting to hire candidates. Let's say we pick the second and third candidates first. To prevent the other recruiter from choosing the same candidate, we can keep these two candidates in a separate room. Another solution is to give each candidate a company-branded T-shirt. When the second recruiter arrives, they can see that the candidate is already selected by the other company based on the T-shirt branding and choose a different candidate.

Similarly, in the table below, instead of using an actual lock, we maintain a "locked" status. When the first customer books a ticket, we change its status from available to locked. When the second customer comes, they check the locked status and are unable to book the same ticket. This is known as a soft lock, where we don't use synchronization to implement a lock; instead, we simply manage the status.



What if after taking the lock the payment never happens ->
We won't keep changing the status repeatedly because if we switch the status from locked to available, we'd have to take a lock again to update it. A better solution is to add a new column called "LockedAt" to the table and record the time when a lock is applied. If someone reserves a ticket but doesn't complete the payment, we can set a fixed time, let's say 10 minutes. If another person tries to book the same ticket after ten minutes have passed (e.g., the first person locked it at 8:12, and the new person attempts at 8:23), we can calculate the difference (8:23 - 8:12 = 11 minutes). Since it's more than the fixed waiting time (10 minutes), we'll allow the new booking, redirect the person to the confirmation page, and update the "LockedAt" column with the new time.

Q: What if the first guy never made the payment.

Let's we allow one person to make the payment for 10 mins.

Show-seats

id	seat_id	show-id	Status (available, locked, booked)	Locked At	
	1	10	~~Available~~ Locked	~~8:12:00~~	8:23:00
	2	10	Available		
	3	10	Available		
	4	10	~~Available~~ Locked	~~8:12:00~~	8:23:00
	5	10	Available		

Another user (1, 4), 8:23:00.

→ (1,4) status

if it's locked,

Current Time - Locked At > 10

8:23:00 − 8:12:00

11:00 > 10.

⇒ Allow him to move to confirmation page.

What needs to be done in service

Booking Service

i/p: userId, showId, showSeatIds.

1. get user details for userId
2. get show details for showId

↙ start trans

3. Set List<ShowSeat> for given Ids
4. check if all of them have status as available.

If not, check if they're locked & lock has expired

5. Mark the status as locked.

Update lockedAt

6. Save List<ShowSeat> in DB.

↙ end transaction

7. create corresponding booking obj / status -> payment pend
8. Return response (Redirect to confirm).

Repositories ->

How to create transactions in spring boot (spring data jpa).

1. Whatever needs to be done in a transaction, extract into a public method.
2. @Transactional(isolationlevel = _____)

How do you interact with db?

Repositories.

But, because of spring data jpa, interactions with db will become very easy.

Student

id	name	psp
1	Keerthi	90

1. establish db connection
2. write query
3. execute query
4. map to your objects
5. get the data

Repositories

interface StudentRepository extends JpaRepository<Student (T), Long (T)>

Student Controller

StudentRepository. _____.save(Student)
_____.findById(123)
_____.count()
_____.findAll()
_____

id	name	psp

Backend LLD-3: Machine Coding - 12: Code BookMyShow 3 - Feb 29

- BookingController
- Booking Service
- Modifying ShowSeat according to our requirement
    - Add lockedAt
    - ShowSeatStatus can be locked as well
- Get user and show details, if they're not present through the corresponding error
- Handle exception
- Get show details
- reserveShowSeats
- Reserveshowseats code
- marking reserveShowSeats method as transaction with isolation level as serailisable
- Create booking object
- Coding the sign up user
- Implement CommandLineRunner to run the code when the application starts
- populating creationTime and updatedTime automatically with annotations
- BCrypt encryption to store the password
- Dependency for storing the password as hash
- (spring-security for bcrypt)
- Code changes for hashing password
- Login user method

==========

Step-by-Step Guide for BookMyShow App Development!

Download it from the video description

https://www.youtube.com/watch?v=et7kLAaOwgk

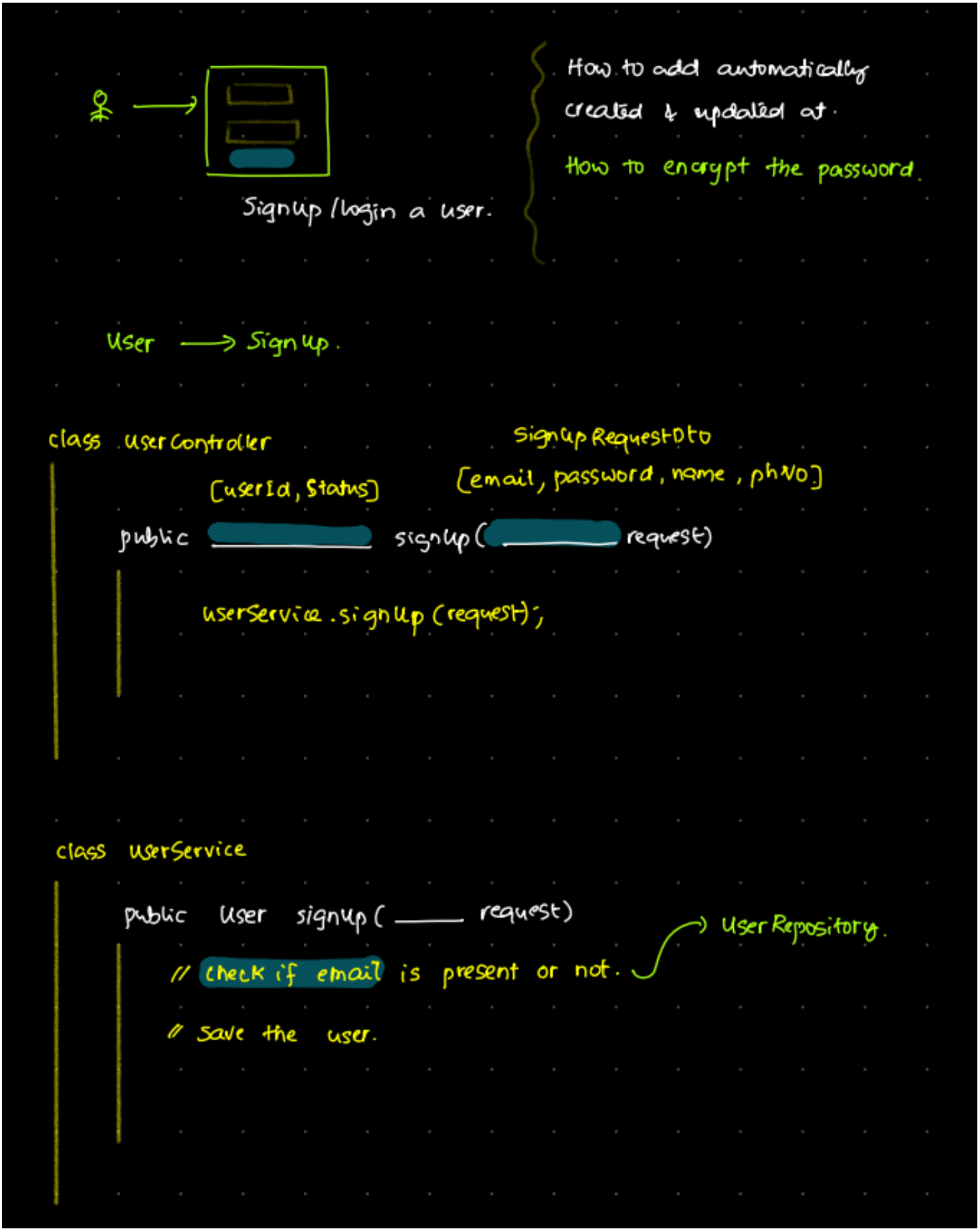
Like | Comment | Subscribe

==========

==========

User Signup and SignIn

==========



interface UserRepository extends JpaRepository<User, Long>

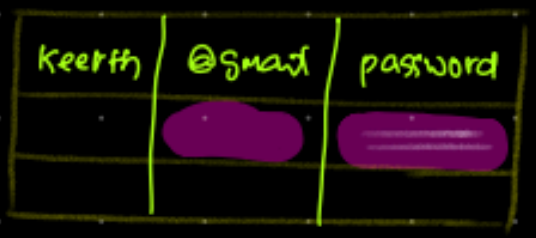
```
Optional<User> findByEmail(String email);
// Select * from user
   where email = "[redacted]";
```

← internal.

BaseModel.

```
createdAt // @CreationTimestamp
updatedAt // @UpdateTimestamp.
```

How are we storing the password?..



pass: Scaler@123.

Scaler@!2A

`!ow secret password.

Save the encrypted password,

Starting a new feature involves creating the UserController, where we'll use the @RestController annotation. write a method signup pass SignupRequestDto as parameter and return type SignupResponseDto

```
@RestController
 public class UserController {
     public SignupResponseDto signup(SignupRequestDto requestDto){

     }
}
```

Next Step: Develop the request and response DTOs.

Request

```
@Getter
@Setter
public class SignupRequestDto {

  private String email;
  private String password;
}
```

Response

```
@Getter
@Setter
@AllArgsConstructor
public class SignupResponseDto {

  private Long userId;
  private ResponseStatus responseStatus;
}
```

Next Step:

1. **Establish the UserService class and implement a signUp method.**
    a. This method will take email and password as strings and return a User.
2. **Set up a UserRepository instance at the top of the signUp method.**
    a. Use @Autowired for automated dependency injection.
3. **In the signUp method, call the UserRepository instance.**
    a. Use the findByEmail method and pass the email.
    b. It will return an Optional<User>, store it in the user variable.
4. **Check user.isPresent().**
    a. If the email is found, throw a new RuntimeException("User already exists") error.
5. **After the condition, save the user using the save method.**
    a. Pass the user object.
6. **Create a new user object.**
    a. Set the email and password.
7. **Pass the new user object to the save method.**

```java
@Service
public class UserService {

  @Autowired
  private UserRepository userRepository;

  public User signUp(String email, String password){
      Optional<User> user = userRepository.findByEmail(email);
      if(user.isPresent()){
          throw new RuntimeException("User is already present");
      }
      User newUser = new User();
      newUser.setEmail(email);

      newUser.setPassword(password);
      return userRepository.save(newUser);
  }

}
```

Next Step: Move to the UserRepository, as findByEmail is not a common method provided by Spring Boot. Write an abstract method in the UserRepository to let Spring Boot create this method.

```java
Optional<User> findByEmail(String email);
```

Next Step: Go to the User class in models and make email and password private strings.

```java
private String email;
private String password;
```

This sequence of steps outlines the process of setting up a new feature, from creating the controller to handling user information in the model. Next Step: Continue with the subsequent tasks in the development process.

Next Step: Now call this signUp method from UserService class into the UserController signup method. first create a instance of UserService and add @Autowired annotation now call signUp method using userService pass email and password from request dto and return new SignupResponse dto by seting user.getId() and status SUCCESS.

```java
@RestController
public class UserController {

  @Autowired
  private UserService userService;

  public SignupResponseDto signup(SignupRequestDto requestDto){
      User user = userService.signUp(requestDto.getEmail(), requestDto.getPassword());
      return new SignupResponseDto(user.getId(), ResponseStatus.SUCCESS);
  }

  public boolean login(String email, String password){
      return userService.login(email, password);
  }
}
```

Next Step: Demo this signup. there are many way to do it like we can create main class and do it from there but To execute it via the command line, navigate to the **DemoApplication** or yourProjectNameApplication class. Implement the **CommandLineRunner** interface, which provides a run method that executes when the application starts. In this class, autowire the UserController. In the run method, create a new SignupRequestDto object, set its email and password, and then invoke userController.signup(), passing the request object to it.

```
@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

  @Autowired
  private UserController userController;

  public static void main(String[] args) {

     SpringApplication.run(DemoApplication.class, args);
  }

  @Override
  public void run(String... args) throws Exception {
     SignupRequestDto signupRequestDto = new SignupRequestDto();
     signupRequestDto.setEmail("prashant@gmail.com");
     signupRequestDto.setPassword("prahsant123");

     userController.signup(signupRequestDto);
  }
}
```

Now run the application and see user table in bokkmyshow database. It is inserted new record

We observe that currently, we are storing the password as a plain string, which is not a secure practice due to potential privacy concerns. To address this, we will store the password in an encrypted form. Additionally, we will introduce the use of the @CreationTimestamp annotation in the BaseModel class. This annotation will be placed above the createAt and modifiedAt fields to automatically save the date in the database.

```
@CreationTimestamp
private Date createdAt;

@UpdateTimestamp
private Date modifiedAt;
```

https://bcrypt-generator.com/ for generate bcrupt password.

We will include a new dependency in the pom file to enable password encryption within the security module.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

If you are unsure about the dependency's name, you can search for "spring security maven" and visit https://mvnrepository.com/artifact/org.springframework.security/spring-security-core. Select the latest version and copy the code snippet. Paste the code into the pom file and click on the refresh button to download the dependency.

Next Step: Navigate to the UserService file. Create a new BCryptPasswordEncoder object in signIp method and utilize the encode method within the setPassword method, passing the password as an argument.

```
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

newUser.setPassword(encoder.encode(password));
```

Note: Decrypting an encrypted password is not feasible every time we need to sign in. For authentication, we will encrypt the provided string and then verify it.

// SignIn —----------

For the sign-in process, let's jump to the **UserController** and introduce a new login method. This method, with a boolean return type, will accept email and password as parameters. Inside this method, invoke userService.login(), passing the email and password, and then return.

```
public boolean login(String email, String password){
   return userService.login(email, password);
}
```

Next Step: Navigate to the UserService, where we will implement the new login method with a boolean return type. This method will accept email and password as parameters. Initially, check if the user is present by retrieving it from the userRepository using findByEmail. Use the isEmpty method to verify its existence. If not present, throw a RuntimeException with the message "Invalid credentials." After this condition, obtain the user using user.get(). Create a BCryptPasswordEncoder and use the encoderObj.matches(password, oldUser.getPassword()) method to validate the password. Return the result of this validation.

```
public boolean login(String email, String password) {
   Optional<User> user = userRepository.findByEmail(email);
   if(user.isEmpty()){
       throw new RuntimeException("Invalid credentials");
   }
   User oldUser = user.get();
   BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
   return encoder.matches(password, oldUser.getPassword());
}
```

Next Step: Following that, proceed to the **DemoApplication**. In the run method, invoke the login method and provide the credentials. Comment out the signup code, and then rerun the application.

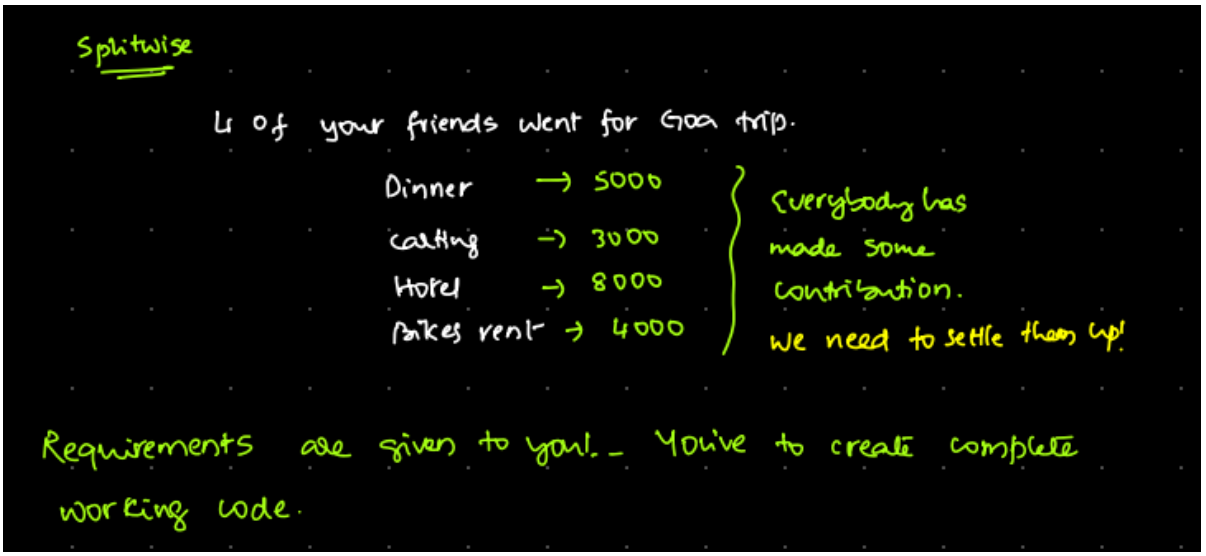
```
@Override
   public void run(String... args) throws Exception {
//     SignupRequestDto signupRequestDto = new SignupRequestDto();
//     signupRequestDto.setEmail("prashant@gmail.com");
//     signupRequestDto.setPassword("prahsant123");
//     userController.signup(signupRequestDto);
      System.out.println("userController.login(\"prashant@gmail.com\", \"prasant@123\") = "
            + userController.login("prashant@gmail.com", "prahsant123"));
   }
```

Backend LLD-3: Machine Coding - 13: Design Splitwise - Mar 02

- **Splitwise Overview**
- **Requirements**
- **What is expense**
- **How to settle up?**
- **Minimum no of transactions**
- **How to calculate amount to pay/recieve**
- **From whom to get/give**

Splitwise Overview

On our trip, four friends did things like eating out, crafting, staying at hotels, and bike riding. Splitwise makes it easy for us to organize and share the costs, making sure everyone pays their fair share transparently.



Requirements

- Users can register and update their profiles.
- A user's profile should contain at least their name, phone number and password
- Users can participate in expenses with other users
- Users can participate in groups.
- To add an expense, a user must specify either the group, or the other users involved in the expense, along with who paid what and who owes what. They must also specify a description for the expense.
- A user can see their total owed amount
- A user can see a history of the expenses they're involved in
- A user can see a history of the expenses made in a group that they're participating in
- Users shouldn't be able to query about groups they are not a member of
- Only the user who has created a group can add/remove members to the group
- Users can request a settle-up The application should show a list of transactions, which when executed will ensure that the user no longer owns or receives money from any other user. Note that this need not settle-up with any other users.
- Users can request a settle-up for any group they're participating in. The application should show a list of transactions, which if executed, will ensure that everyone

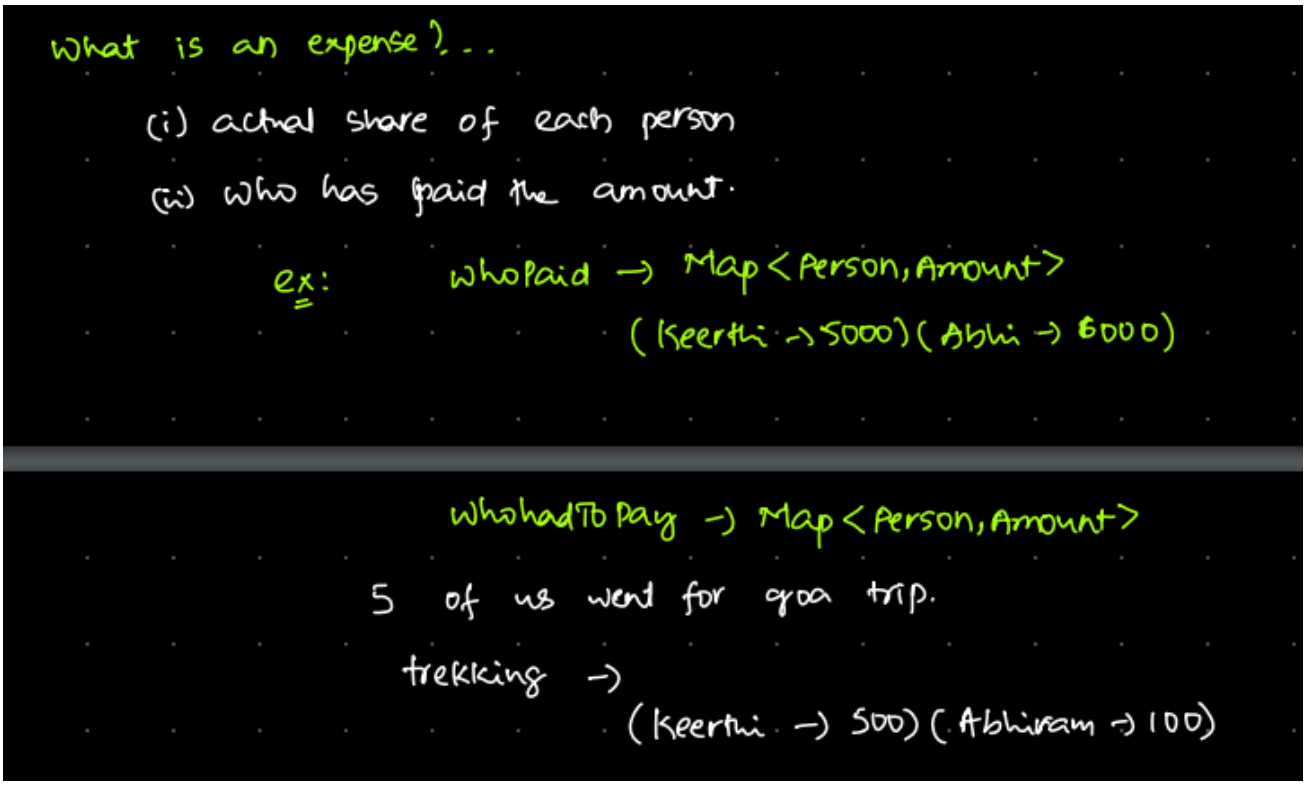
participating in the group is settled up (owes a net of 0 Rs). Note that it will only_deal with the expenses made inside that group. Expenses outside the group need not be settled.

- When settling a group, we should try to minimize the number of transactions that the group members should make to settle up.

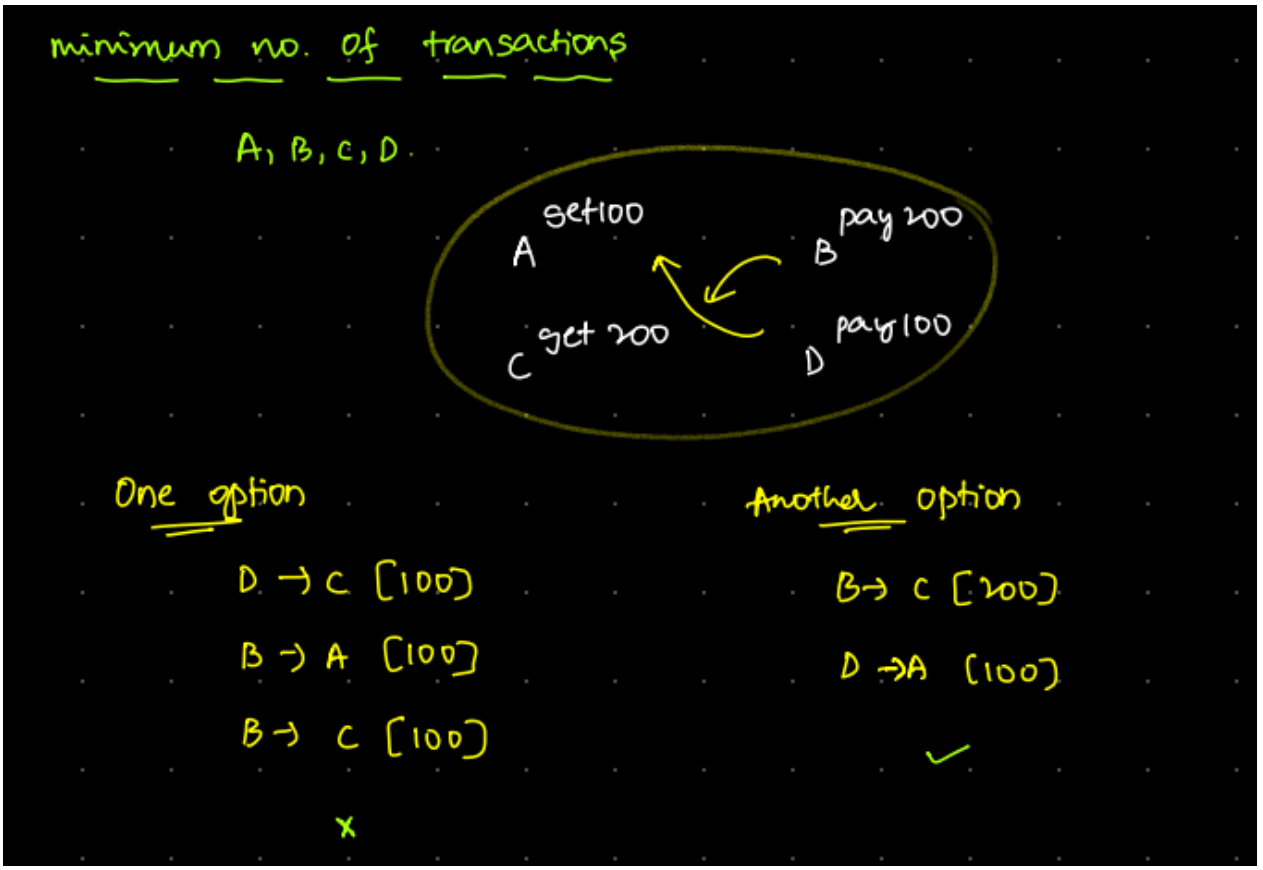
  Note: All tests will be performed in one go. The application doesn't need to persist data between runs.

The existing requirement provides clear guidance. When a requirement is given, the visual approach is unnecessary. Instead, we should meticulously read the requirement line by line, identifying nouns. Following this, we can construct a class diagram and proceed with module creation.

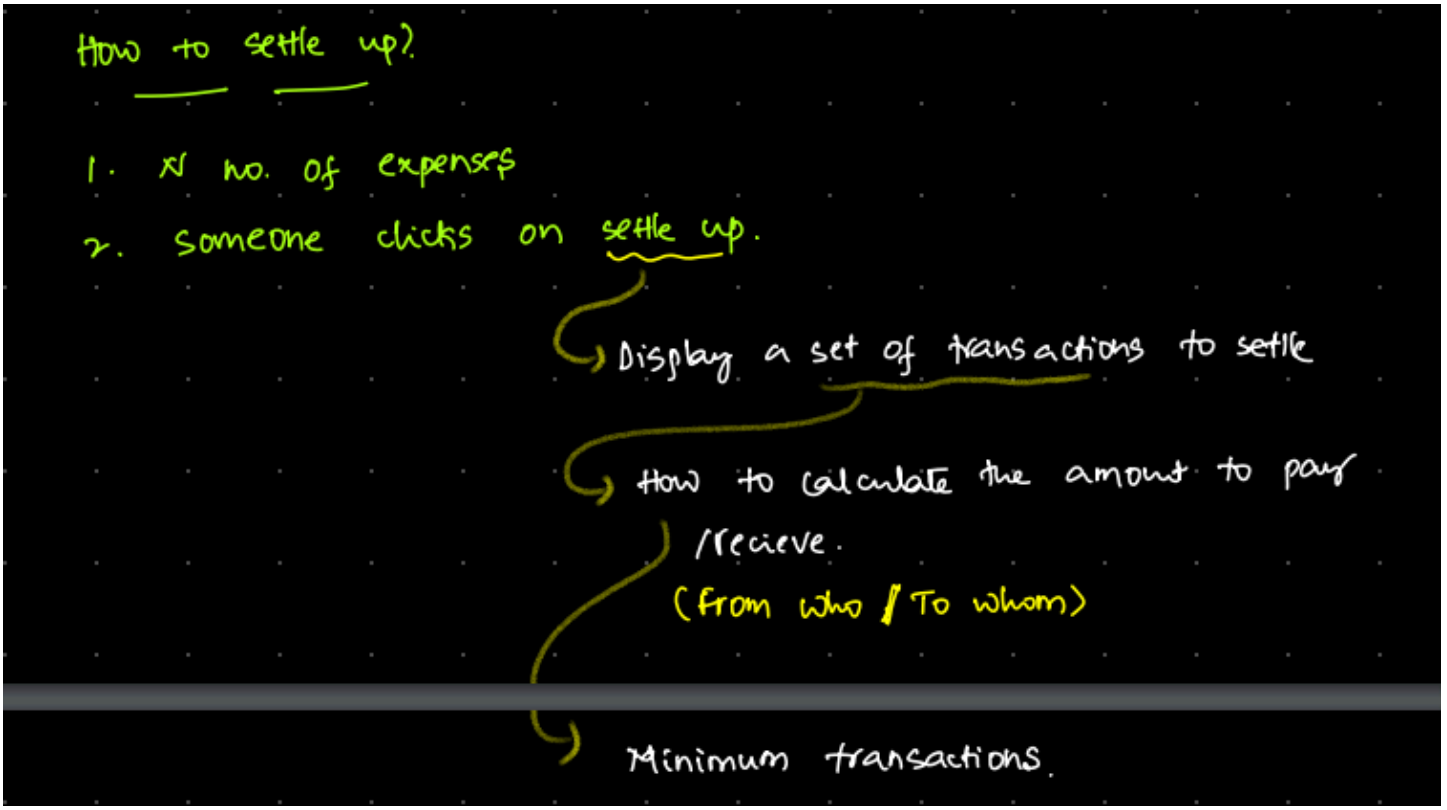
What are expenses? -> Expenses represent the portion of real money spent by each individual. For instance, in the scenario where five people went on a trip, but only two of them went tracking, we won't distribute those costs among all five. Instead, these expenses are specific to the two individuals involved, and they will only be distributed among them.

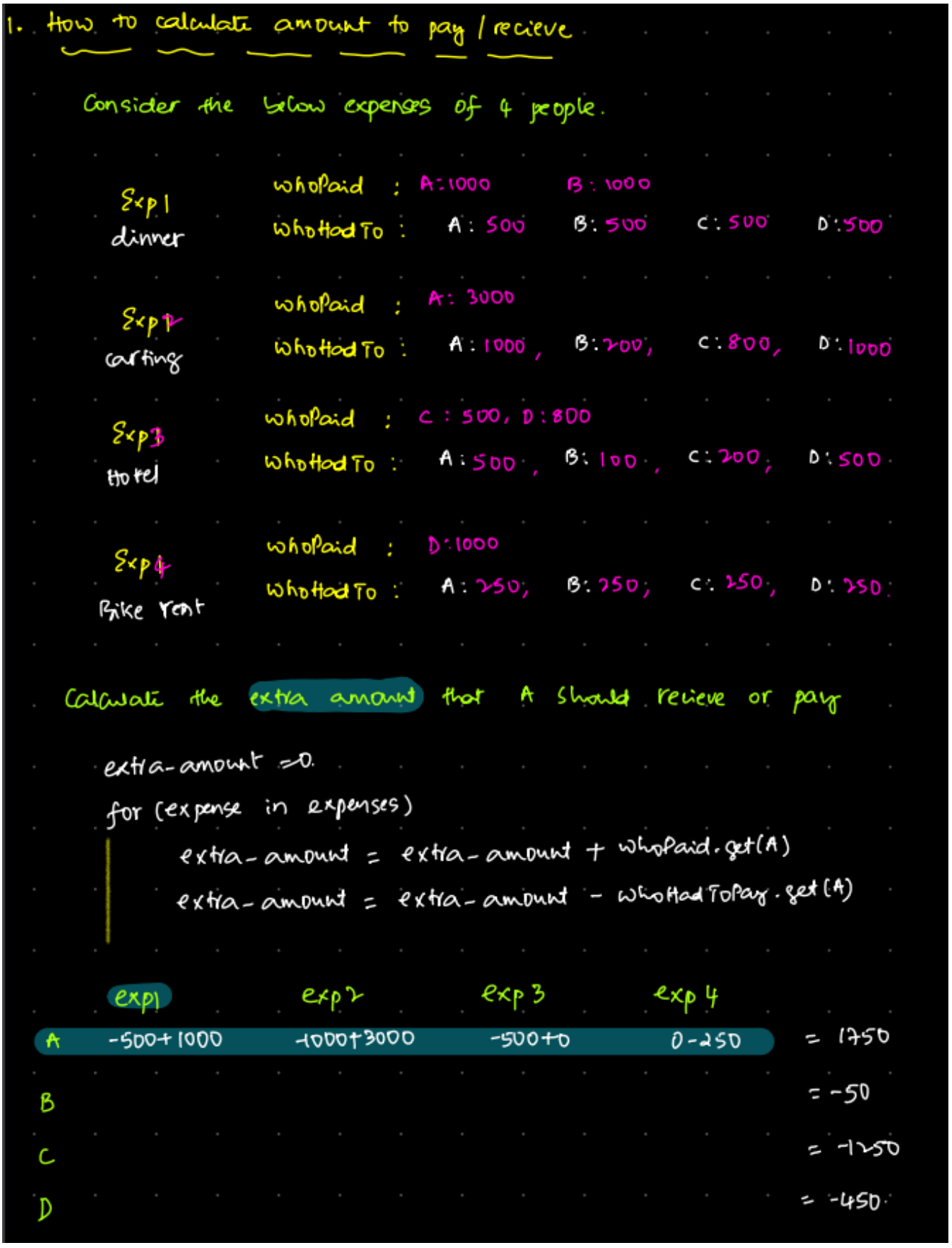


Minimum No of Transactions -> Since there can be various ways to resolve payments, the first option involves three transactions, while an alternative solution requires only two transactions, indicating that the latter option minimizes the number of transactions. To address this, we will develop multiple strategies.



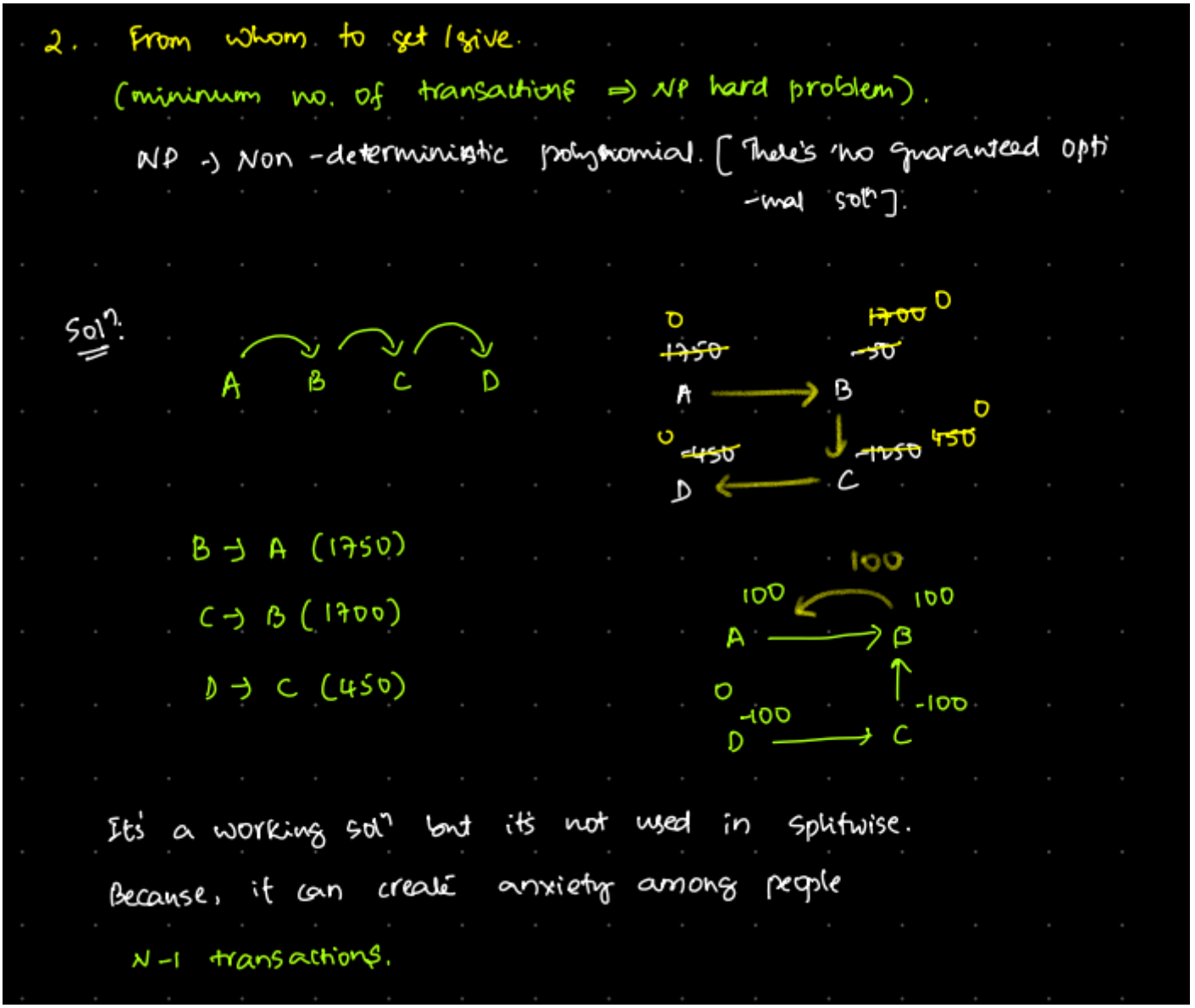
How to settle up?





Simply put, by totaling the amounts given by person A and subtracting the shared expenses, we can determine whether A needs to receive money or pay extra. For example, if A gave 1000 the first time but their share was 500, it means they have to pay 500. Similarly, if A gave 3000 the second time but their share was -1000, and they gave nothing the third time with a share of 500, and gave nothing again the fourth time with a share of 250, the calculation would show that A needs to receive a total of 1750 from the other friends.

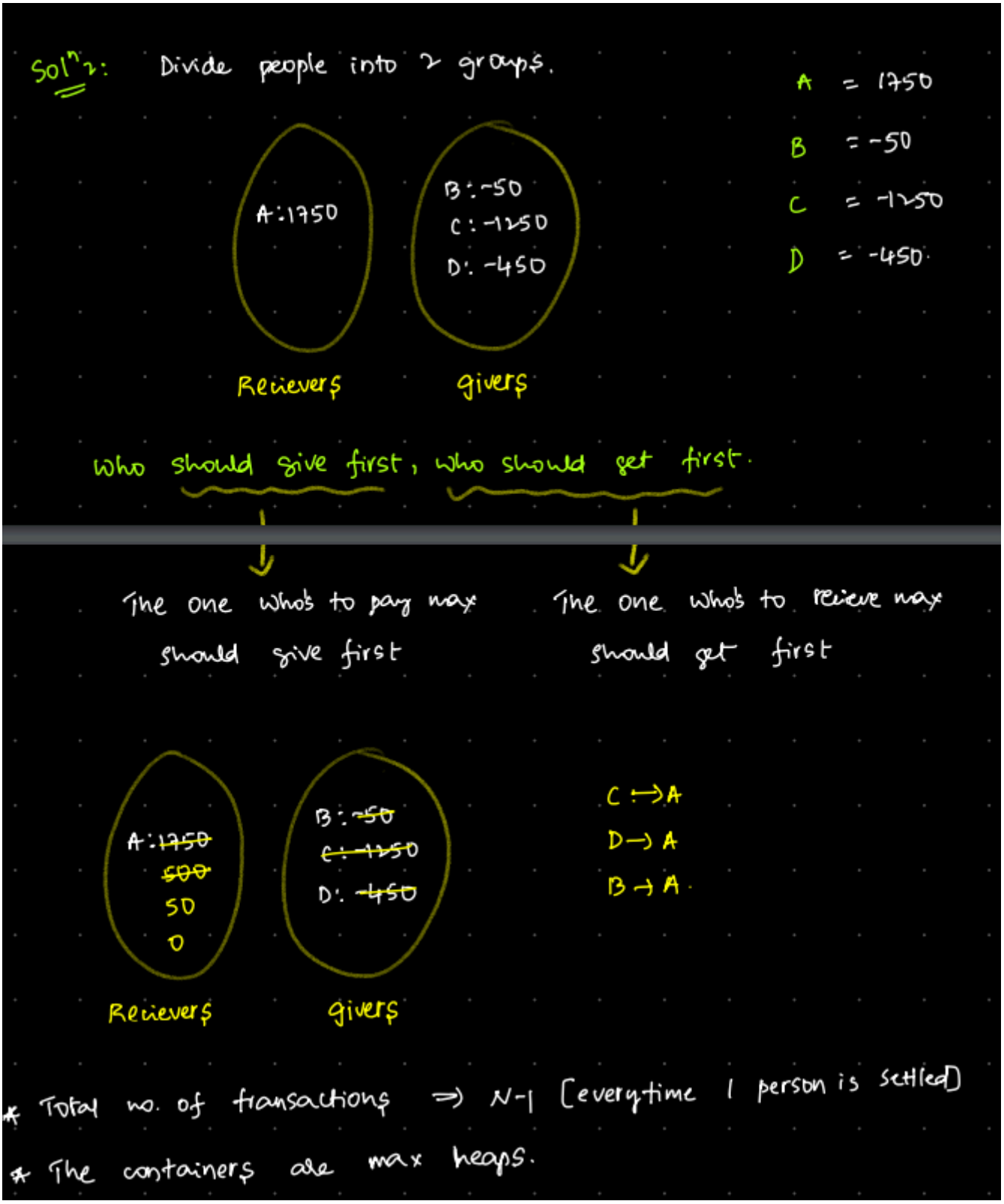
The problem of finding the minimum number of transactions is referred to as an **NP-Hard problem**. **NP stands** for **Non-deterministic Polynomial**. In this type of problem, the solution space grows exponentially, making it challenging to guarantee an optimal solution. While an optimal solution may be achievable in specific cases, it is not guaranteed across all instances.



In the above scenario, the initial step to settle the amount involves B giving 1750 to A. Consequently, B has a pending payment of -50, which has already been covered by the 1750 given to A. Now, B needs to receive 1700, and this amount is settled by C. C, in turn, has to receive 1250, and once settled, C pays 1700 to B. After this, B has to receive 450, and C settles this by giving 450, as D is required to pay this amount. Ultimately, the settlement process concludes with all amounts settled.

In reality, this approach is not practical for everyday situations. If I only need to pay 50, it doesn't make sense to pay extra and wait for someone else to settle the remaining amount.

In the second strategy, we divide it into two parts: recipients and givers. Initially, we prioritize the recipient with the highest amount in the first part, and in the second part, we prioritize the giver with the maximum amount. To address this, we employ a priority queue, ensuring that the maximum amounts are at the top. The top giver attempts to settle with the top recipient; if successful, the recipient is removed from the first part. If there's a remaining balance, the recipient is reinserted into the first part. Similarly, if the giver still has a pending amount, they are reinserted into the second part with the remaining balance. This process continues until both recipient and giver lists are empty.



C: -1250

A: ~~1750~~ -500

A is not settled yet.

Push A back to heap.

D: -450

A: 500 -50

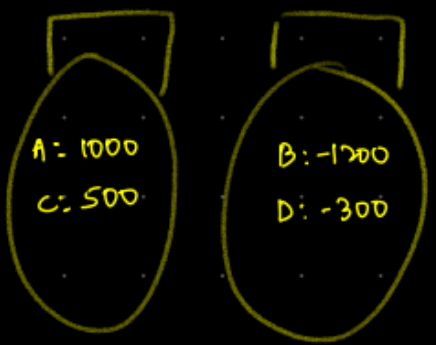
A is not settled yet.

Push A back to heap.

B: -50

A: -50

Heaps are empty, stop here.



B: -1200

A: ~~1000~~ 0.

TC: NlogN.

Total transactions are (N-1).

HLD:

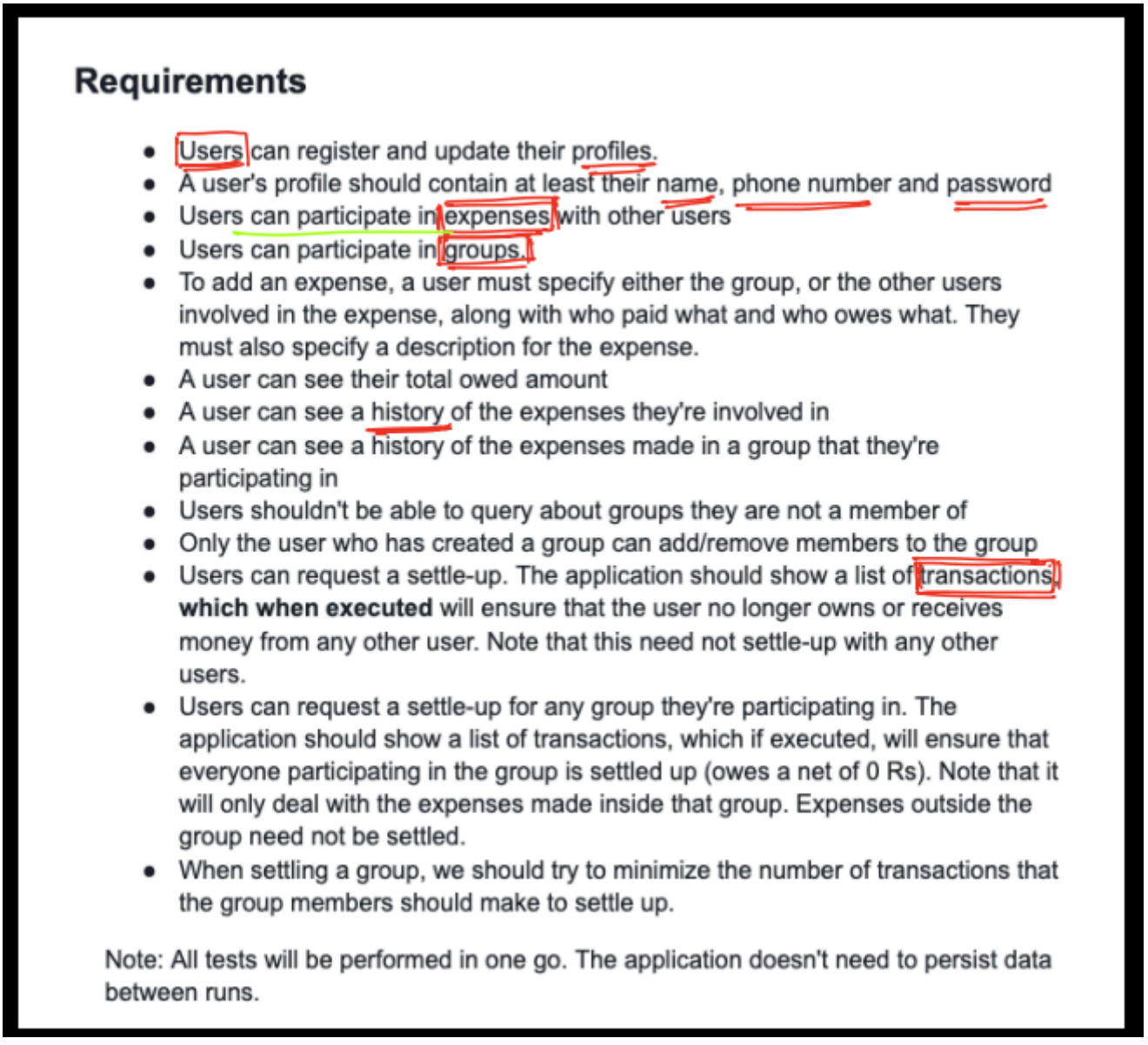
1. identifying nouns
2. Create class dia / schema dia
3. create models
4. write the code for settleup strategy.

(Heap settleup strategy).

Backend LLD-3: Machine Coding - 14: Code Splitwise 1 - Mar 05

- **Going through requirements to find out the noun**
- **Will transactions become a table/model ?**
- **What happens when you click on settle up ?**
- **How to handle completed transactions**
- **Introducing dummy transaction**
- **Revise dummy transaction concept**
- **Class diagram**
- **UserExpense table**
- **Easiest way to handle commands (Monster method)**
- **Introducing Command interface**
- **Introducing commandRegistry/commandExecutor**

Going through requirements to find out the noun ->
I will go through each requirement one by one and identify nouns with extra details, such as attributes.

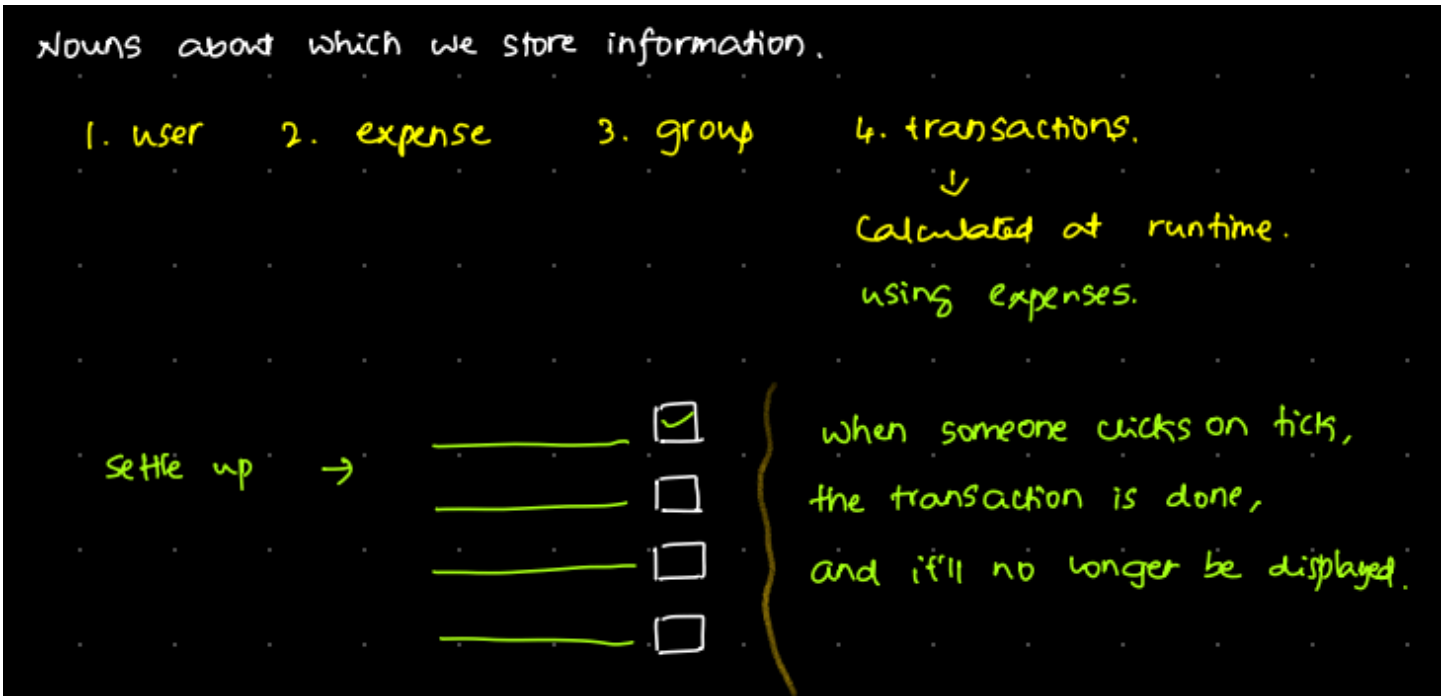


Requirements

- Users can register and update their profiles.
- A user's profile should contain at least their name, phone number and password
- Users can participate in expenses with other users
- Users can participate in groups.
- To add an expense, a user must specify either the group, or the other users involved in the expense, along with who paid what and who owes what. They must also specify a description for the expense.
- A user can see their total owed amount
- A user can see a history of the expenses they're involved in
- A user can see a history of the expenses made in a group that they're participating in
- Users shouldn't be able to query about groups they are not a member of
- Only the user who has created a group can add/remove members to the group
- Users can request a settle-up. The application should show a list of transactions, **which when executed** will ensure that the user no longer owns or receives money from any other user. Note that this need not settle-up with any other users.
- Users can request a settle-up for any group they're participating in. The application should show a list of transactions, which if executed, will ensure that everyone participating in the group is settled up (owes a net of 0 Rs). Note that it will only deal with the expenses made inside that group. Expenses outside the group need not be settled.
- When settling a group, we should try to minimize the number of transactions that the group members should make to settle up.

Note: All tests will be performed in one go. The application doesn't need to persist data between runs.

Will transactions become a table/model ?
The Transactions table is unnecessary since we need to calculate it dynamically from the Expenses class during runtime.



What happens when you click on settle up ?

According to the requirements, there are two approaches to handle removing completed transactions from the list upon the next refresh. The first approach involves creating another class named TransactionDone and adding completed transactions to it. However, this approach is considered less optimal as it introduces an additional class to manage.

Introducing dummy transaction

A more efficient solution is the second option. In this approach, when settling up, a dummy expense is created in the transaction table to counterbalance other transactions. This dummy expense serves as a placeholder, and during subsequent calculations, when comparing with the original transactions, it results in the net effect of making that transaction amount zero.

The recommended solution is to use the second option, as it avoids the need for managing an extra class and achieves the desired outcome more effectively.

When someone clicks on 'tick'.

Option1: Create a class called 'DoneTransactions', and add the transaction to this.

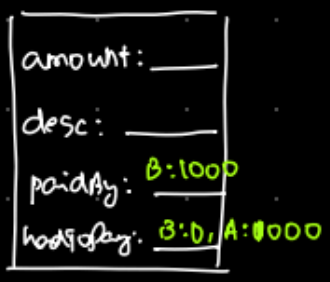
For a group, you will calculate List of transactions, filter the done transactions.

Not so great!..

Option2: When a transaction is done, you try to add a dummy expense.

Such that when we calculate List of transactions next time, we don't get the done transaction.

Ex:

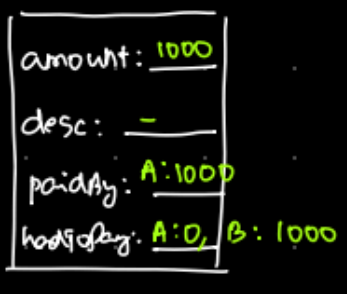


expenses.

Transactions.

A -> B : 1000 ☑ {B has paid 1k extra}

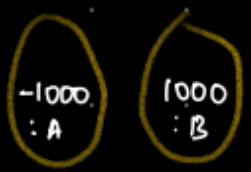
We add a dummy expense.



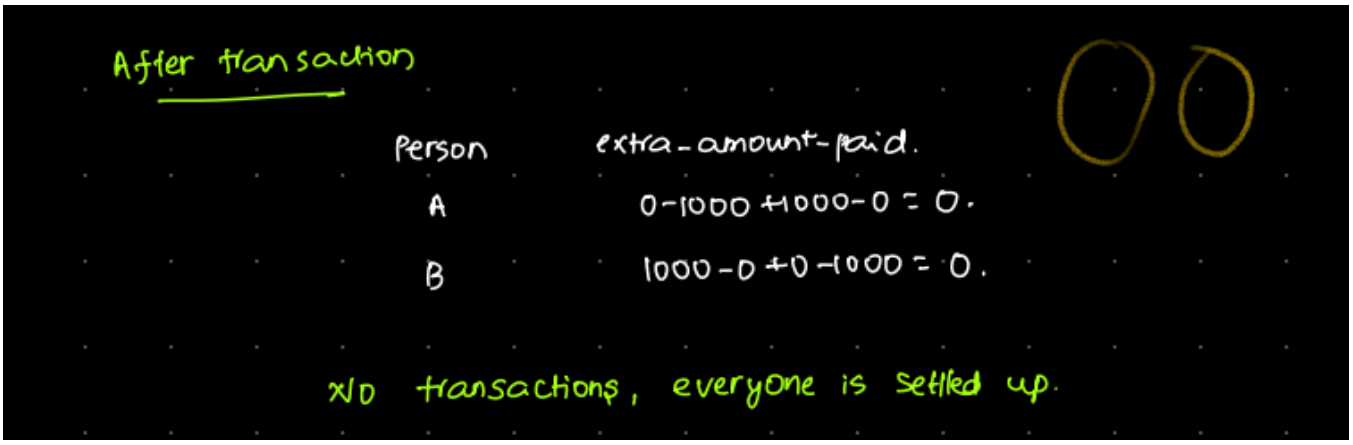
Revise dummy transaction concept

Before transaction:

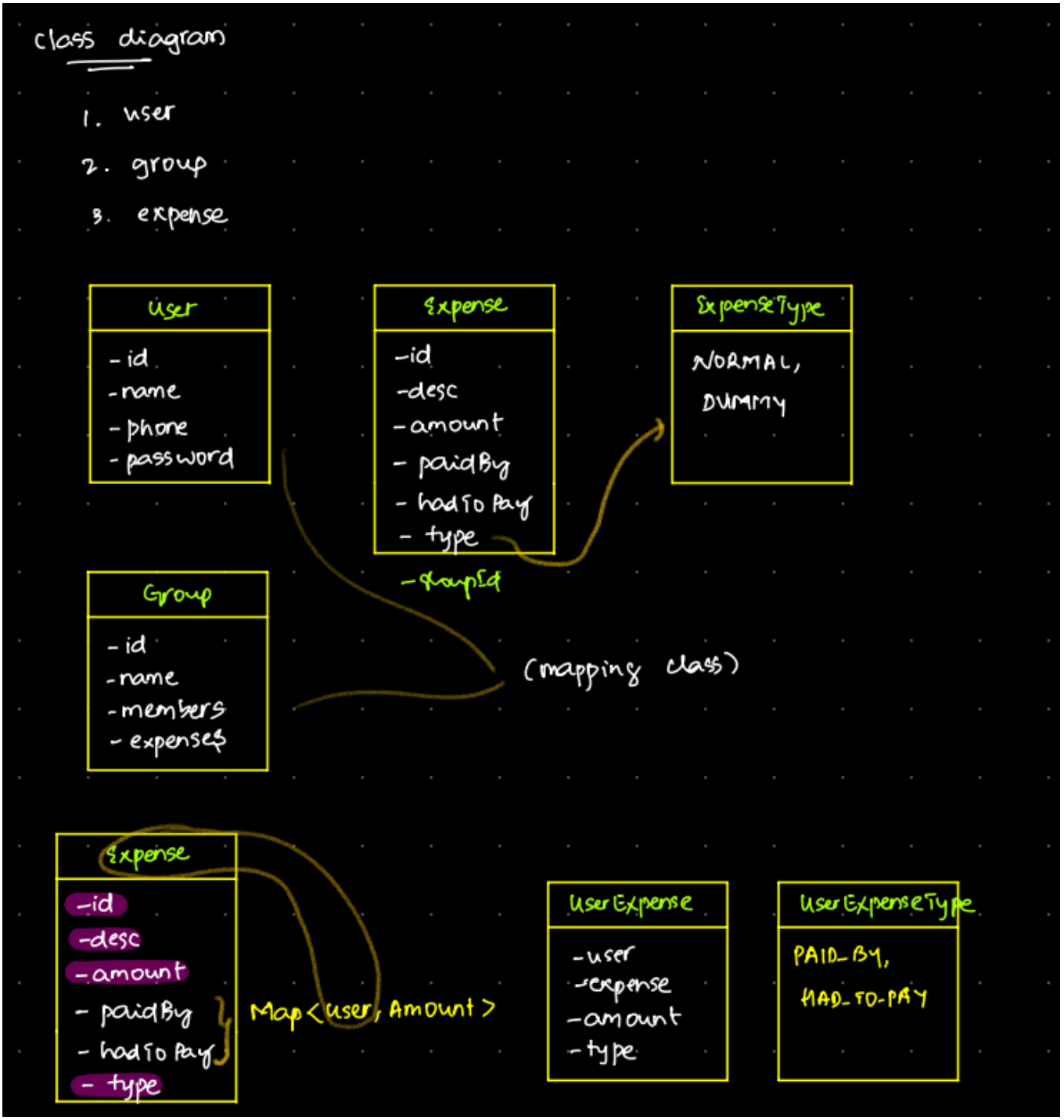
Person	extra_amount_paid.
A	0 - 1000 = -1000
B	1000 - 0 = 1000



A -> B : 1000 ☑

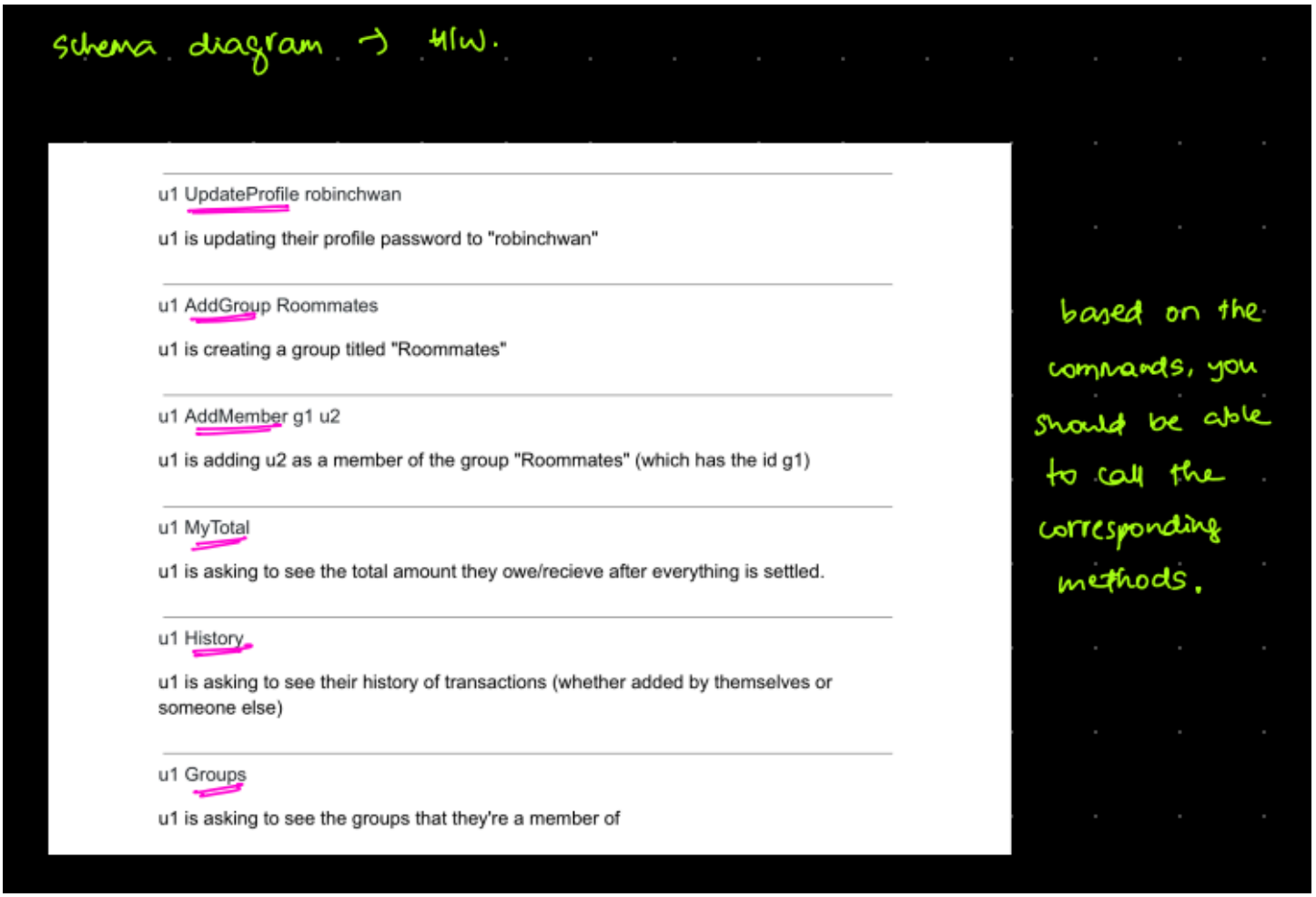


Class diagram

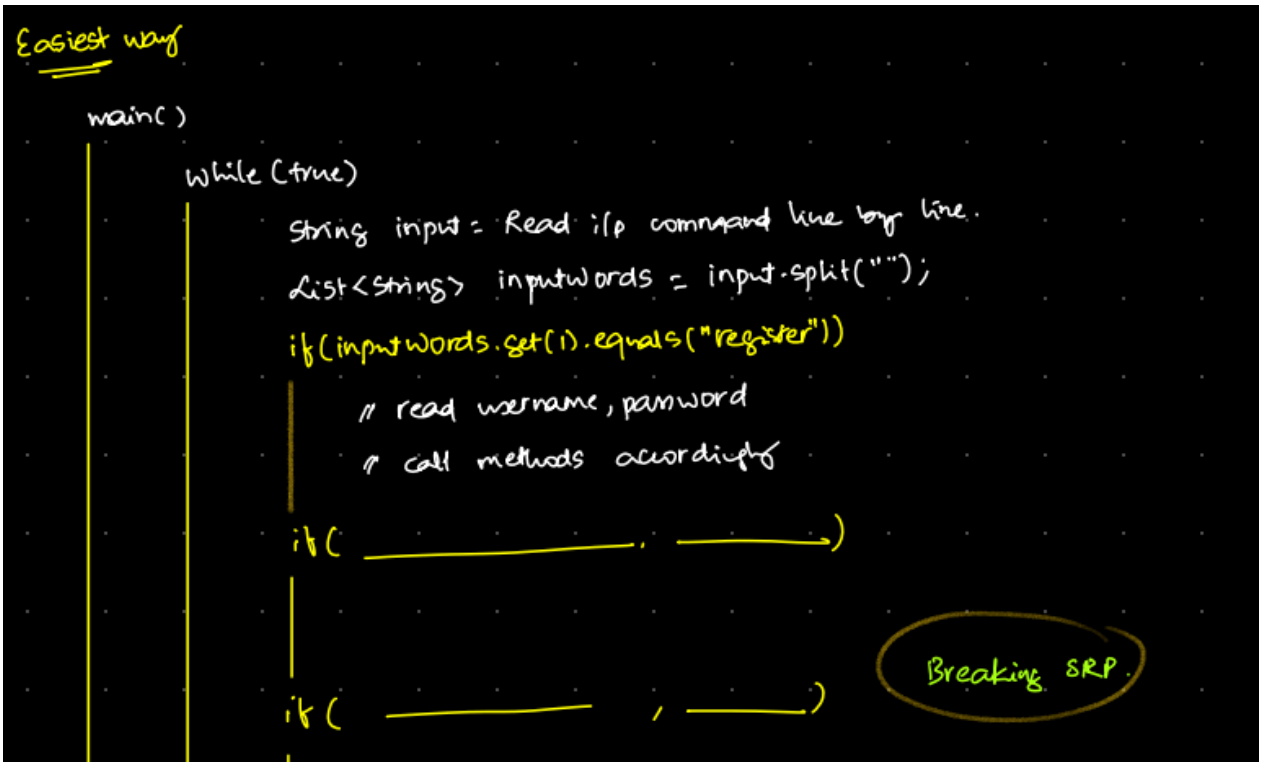


UserExpense table

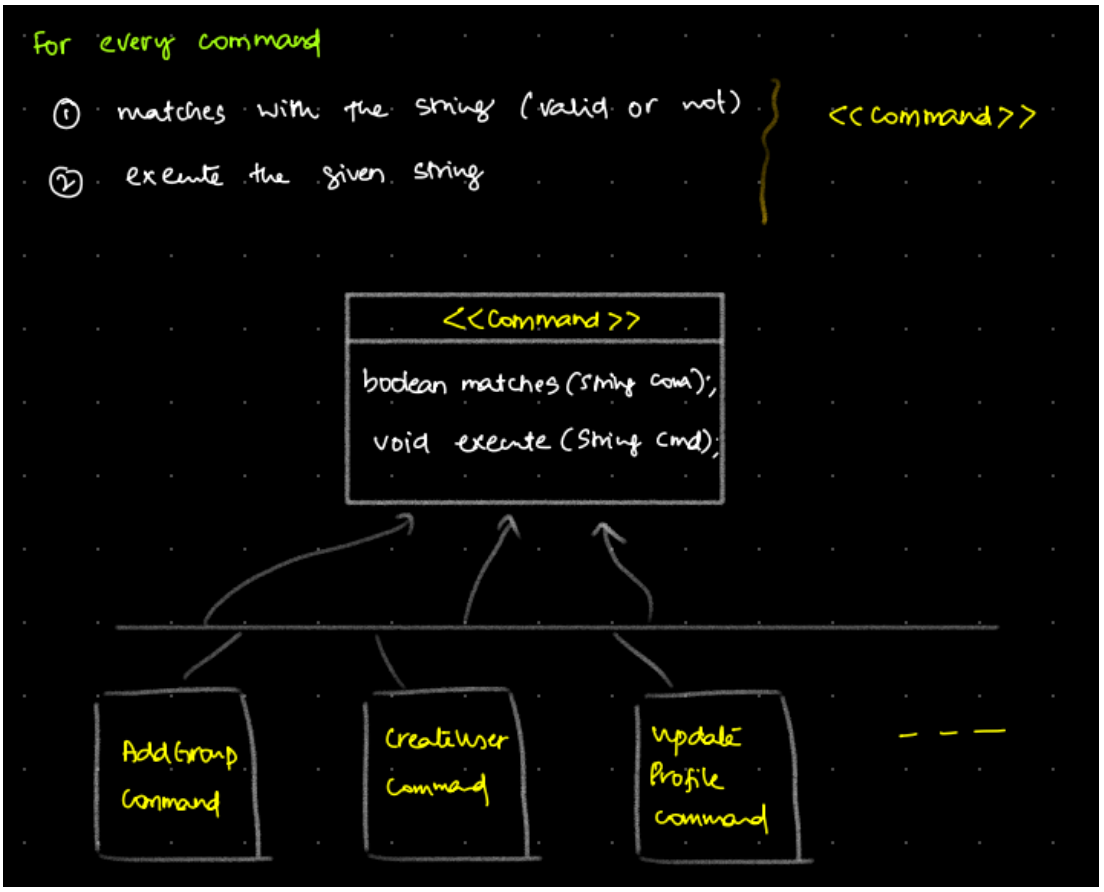
After checking the requirements, we found three categories: User, Expenses, and Group. Initially, we used a list of users like 'paidBy' to connect expenses, but realized this created problems with tight connections. So, we added a class named 'UserExpenses' to make the design more flexible and scalable. Similarly, for groups, we avoided directly linking them to users and vice versa to reduce connection problems. Instead, we added a mapping class called 'UserGroup.' To distinguish between regular and dummy expenses, and types within the 'UserExpenses' class (such as 'paidBy' and 'hadToPay'), we introduced an enum called 'ExpensesType' and another enum named 'UserExpensesType.'



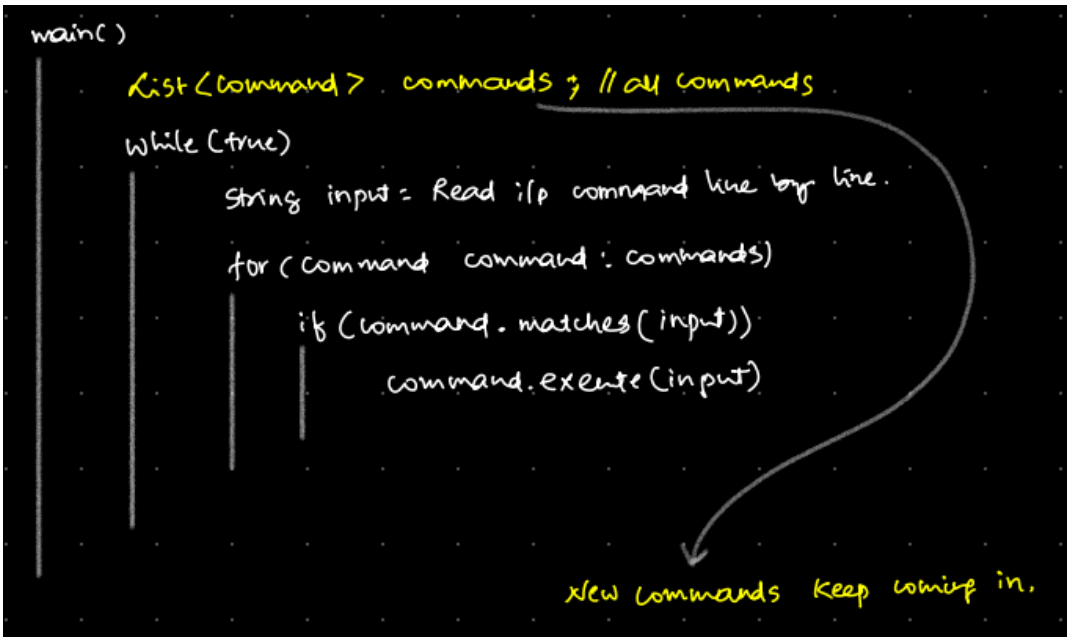
The simplest way to manage commands (Monster method) is by reading the file line by line using a scanner in a while loop. Within the loop, use an if condition to check for action keywords and perform the corresponding tasks. However, this approach violates the Single Responsibility Principle (SRP) because handling all conditions becomes a monolithic method. This is not ideal, as it lacks scalability and can lead to future bugs.



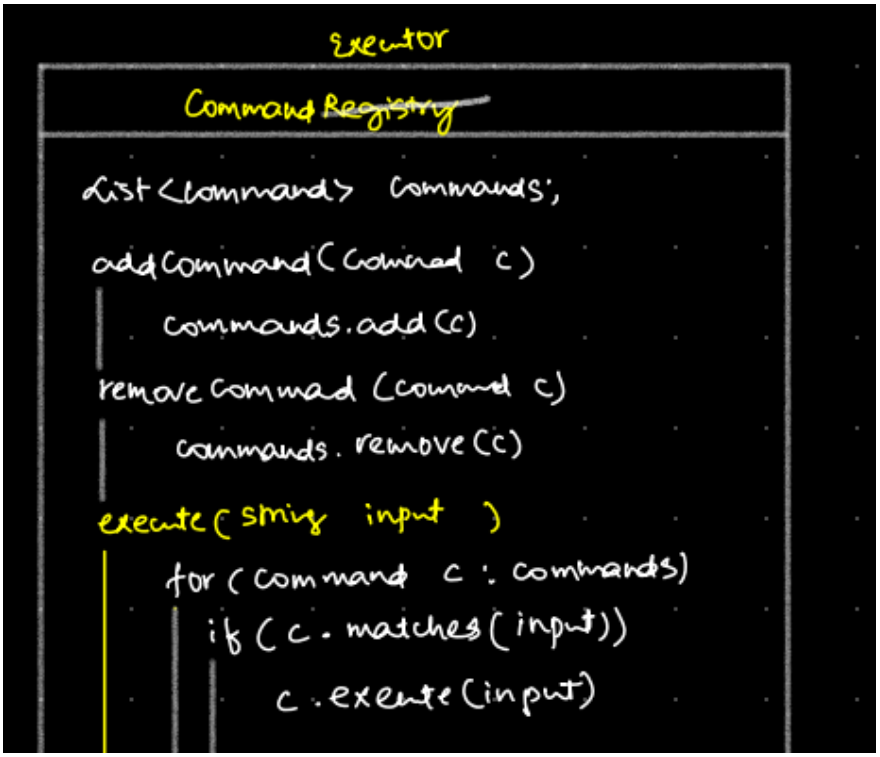
In the monster method, for each command (condition), we currently perform two tasks: matching the string and executing the task. It would be a good idea to separate these tasks into a new Command class. Since we perform similar tasks with different types, it's beneficial to create an interface for these commands, like addGroup command or register command. Next, we'll design a Command interface with two methods: 'matches,' which returns a boolean and takes a parameter cmd of string type, and 'execute,' which returns void and accepts a parameter cmd as a string.



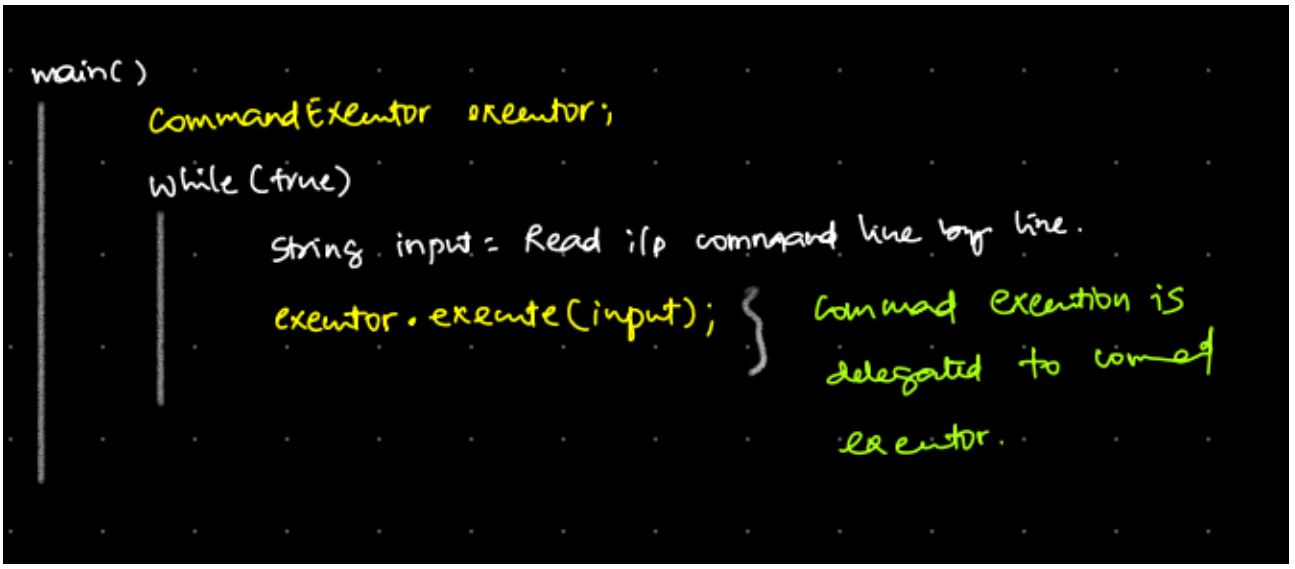
To refactor the Monster method, begin by removing all conditions and retaining the while loop. Now, create a list of commands. Begin reading the command file within the loop using a scanner. Inside this loop, add another loop to iterate through all commands. Check each command in the list, and if the input matches, execute that command; otherwise, proceed to the next one.



There's another issue with the above approach: new commands keep coming regularly, and updating the list on the client side isn't a good idea. A better approach is to create a registry (Registry is the class where we create a list). While adding a new command, we check if it's available in the list. If it is, we return the existing one; otherwise, we create a new one. To implement this, we expose methods like addCommand and removeCommand. Since we have all the commands in the registry, it's a good move to shift command-related logic, including the foreach loop from the main method, to here and create a new execute method. Since we're now handling multiple tasks, it's not accurate to call it a registry. Let's rename it to CommandExecutor.



Now, let's make changes in the main file. Create a CommandExecutor instance named 'executor,' and within the while loop, read the file and execute 'executor.execute(input).' The remaining logic has already been addressed in the CommandExecutor.



Backend LLD-3: Machine Coding - 15: Code Splitwise 2 - Mar 07

- **New project and code models**
- **Repositories**
- **User Controller and service**
- **Understanding how to settle an user**
- **Steps to do inside settleupUser method**
- **Understanding the steps with example**
- **Continue coding the steps in service**
- **Strategy**
- **Filter user transactions**
- **HeapSettleUpStrategy (homework, pasting the code for reference)**
- **Demo, preparing the data**
- **CommandExecutor**

==========

Comming soon on

https://www.youtube.com/@GeekySanjay

Like | Comment | Subscribe

Thanks You

==========