Lecture 37 : Build Chess Game

classmate
Date
Page

Requirement

- → We can have multiple users playing chess at the same time.
- → Score based match making algorithm
- → We have standard chess rules but can be scalable.
- → Users within a match can also send/receive messages
- → Users can quit the match in between.

Happy Flow

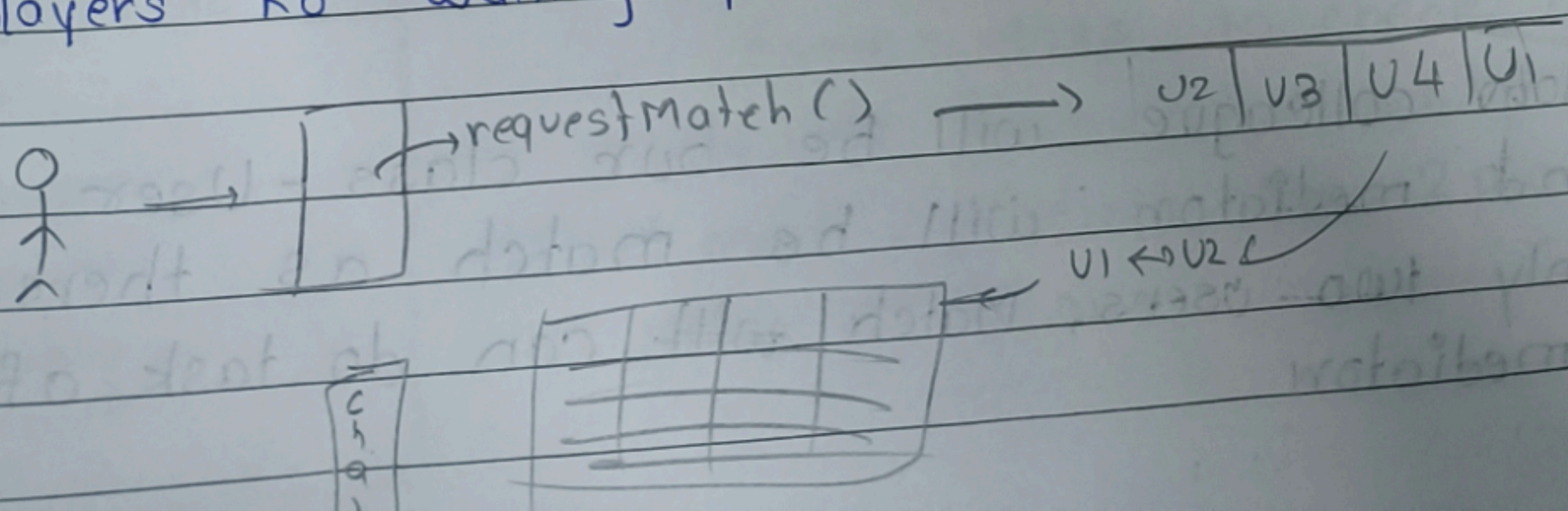
(beginner level)

① User1 comes to our platform and then requestMatch() and adds user1 in waiting queue.

② In waiting queue there are user2 (↓ beginner level), user3 (↓ intermediate level), user4 (↓ advance level) waiting for getting best Match.

③ Then we matched user1 and user2 and entered them into room / UI of chessboard

④ Ab inmein se koi ek player jitega / draw hoga yaa koi chatroom leave kar dega toh hum wapas se players ko waiting queue mein le jayege.



UML Diagram of Chess Game

① We will be using top-down approach to build this game.

② Let's make first main orchestra class - GameManager class which is a singleton class.

③ Now components in class -
- Every game will have unique id as there might be multiple games running parallely so to store them we will have - map<String, Match> allMatches

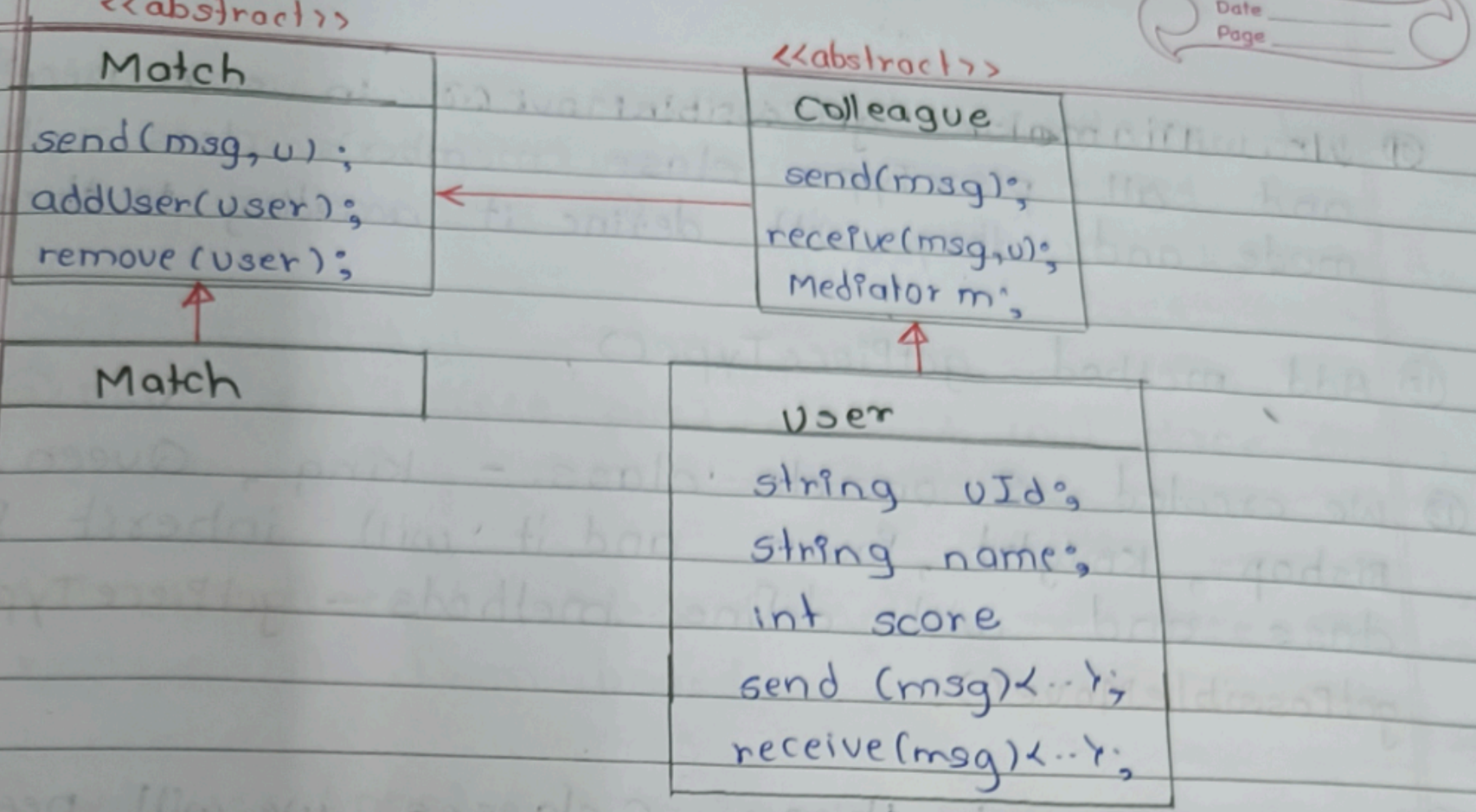
④ Now we will create Match class first

Match		User
String matchId; User bU, wU;	→	String uId; String name; int score;

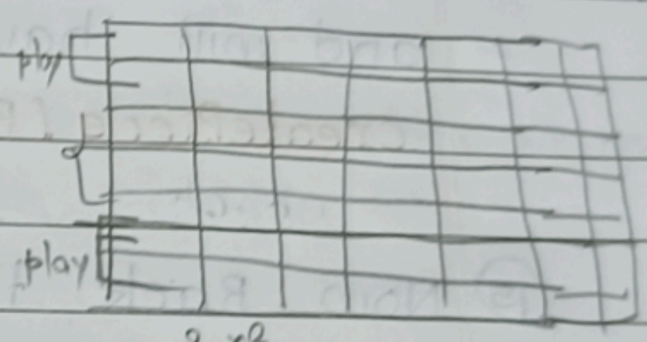
let's create user class first

⑤ Now in our requirement no. ④ we want users to send / receive message. For implementing it we will be using Mediator pattern.
- Mediator pattern needs two thing ① Collegue - who wants to communicate ② Mediator - which will help them to communicate.

⑥ Now colleague will be our class - User and mediator will be match as there are only two users, Match ~~will~~ can do task of mediator



⑦ Now, in match we will need a Board on which we will play games. (8×8 board)



⑧ Board will be of 8×8 so we will create a 2D vector and in that vector each place will have pieces - like pawn, [illegible], king, [illegible], etc.

⑨ So let's create chess piece - Piece will have color so lets create enum name color Piece class will also have type of piece like whether it is pawn, knight, etc

<<enum>>

Color
BLACK, WHITE

PieceType
king;
Queen;
Rook
Bishop, knight
PAWN.

⑩ we will add bool isMoved in Piece class to check whether it is his first move as pawn & in first move can move 2 place but later only one.

(11) We will make getPossibleMove() in our Piece class and all pieceType ~~class~~ member's class will be made and they will define it according to them

(12) Add method getPieceType()

(13) We created 6 concrete class - King, Queen, Rook, Bishop, Knight, Pawn and it will inherit Piece class and will define methods - getPieceType(), getPossibleMove().

(14) To manage all these 6 classes we will need a PieceFactory and it's task is to create Piece and will have method -
createPiece(PieceType pt, Color c){...};

(15) Now Back to Board class - we will create a map <Position, Piece> PiecePosition;
↳ can be replaced by pair<int,int> to get actual location but it will be hard to work in code due to it.first & .second ∴ we will create separate class.

(16) Let's create Position

Position
int row;
int col;
isValid(){...}

it will check if pos it move to isValid or not as it ~~can be~~ move in wrong dorn or something.

(17) Back to Board -
① initialize(){...}: whenever board is made it is called (constructor like work).

(19) Now, we will keep our Board class dumb as it shouldn't know if it is valid move or rules to move a piece. we will use Rule strategy.

(19) Continue Board -

(2) placePiece (Piece, pos) {..} : It will place pieces on position using map → PiecePosition.

(3) removePiece (Pos) {..} : Capture a piece ~~fr~~ and remove it from board.

(4) getPiece (pos) {..} : It will getPiece from posrtion using map.

(5) isOccupied (pos) {..} : It will check whether ~~it~~ pos has piece already or not

(6) movePiece (from, to) {..} : It will move piece from position, to next position.

(7) display () {...} : display current state of board.

(20) Now we will need Rules
- we will create abstract class - chessRule
It will have methods -
(1) isValidMove (Move m, Board b) : To check valid move we create Move class as it will need too much parameter like pieceType, current pos, ~~to~~ color, etc.

Move
position from;
Position to;
Piece piece;
Piece captured;

(2) isInCheck (color c, Board b) :
It will check if any color player is in check position or not.

③ isCheckMate(color, board): King is in check position and there are no ways to be safe our king.

④ isstaleMate(color, board): King is in safe position but wherever it will move it will get check.

⑤ wouldCauseCheck(color c, Move m, Board b): if there is any move if it takes in that position it will get check.

(22) Now we will create concrete class where we will define all this methods

(22) Back to Match class -
We will create a enum class called GameStatus to checks ~~wh~~ the status has game.

(23) Our Match class is also a mediator class so it should store history of our chats so we will ~~h~~have vector for it - ~~vector<Initi~~
vector<Message> history.

(24) Let's create ~~history~~ Message class

Message
string senderId; string content; timestamp

(25) We will also have movehistory vector in our Match class
vector<Move> moh;

(26) Methods in Match class -
① makeMove(from, to, user){..}: create Move object and will then call rule class to check move valid & other folbwup methods.

② exitGame (user) {...} : Game is still going on but player left

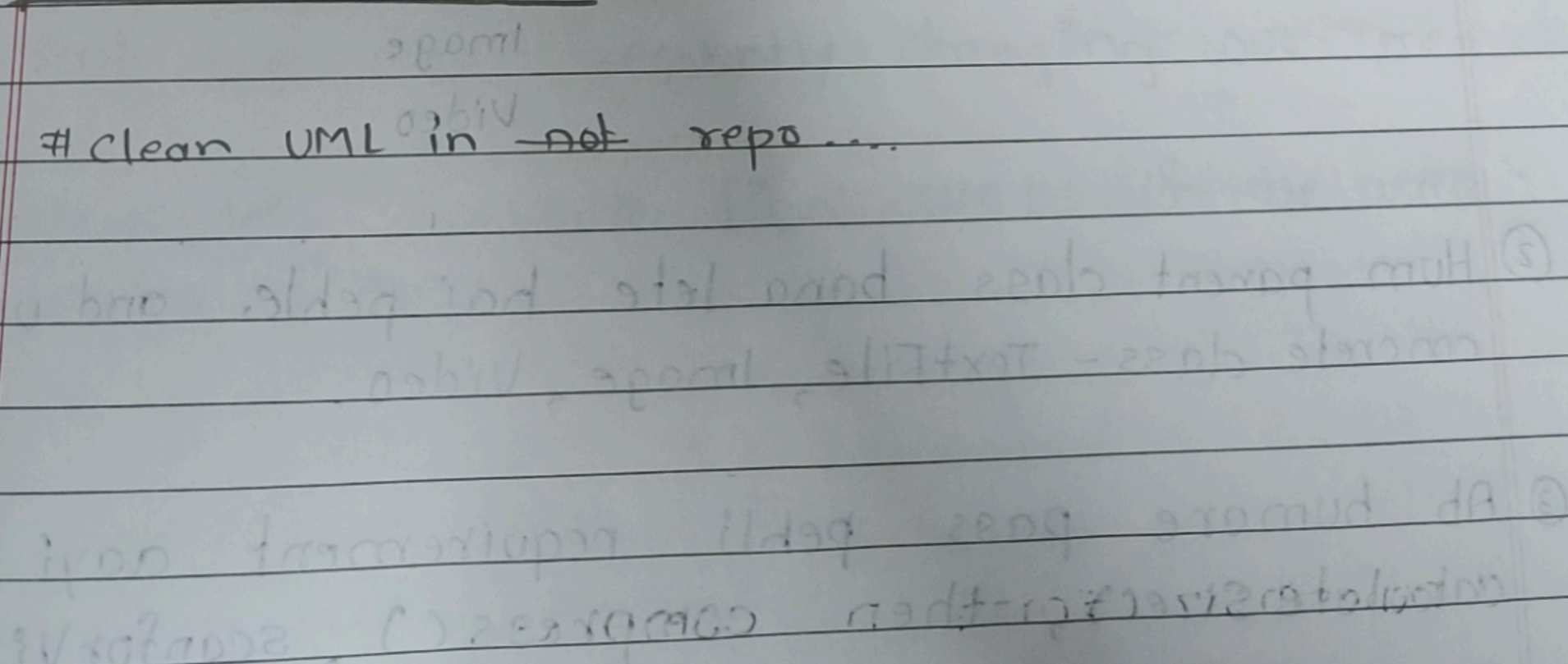
③ endGame (user, rsn) {..} : Game ends & will print which user won and reason why other lost

④ send (msg , u) {...} :

⑤ addUser (user) {..} :

} Mediator class work

㉗ Everything is completed only last task is remained i.e. GameManager class



Clean UML in ~~not~~ repo ...