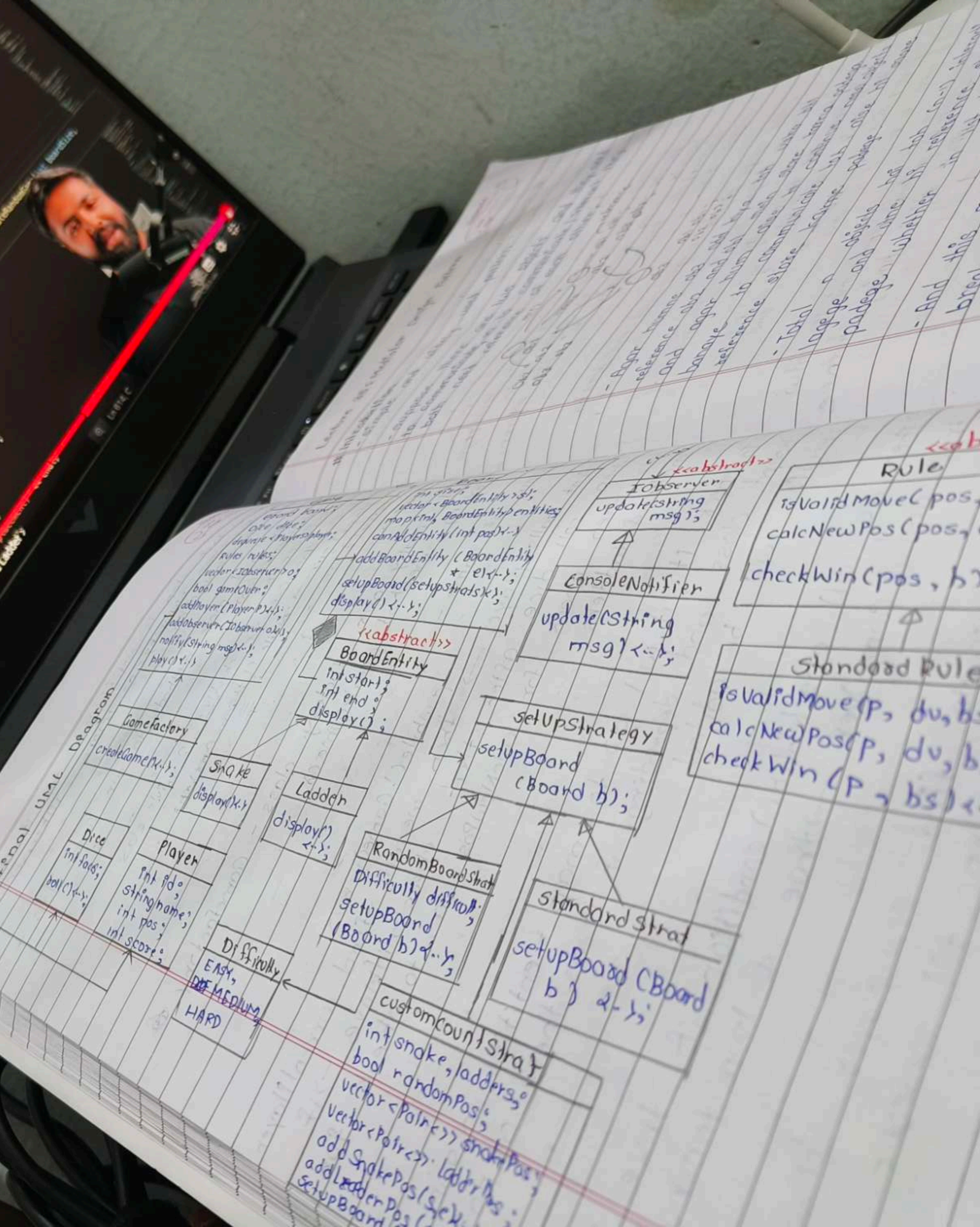

d Snake and Ladder Game | UML + Code | System Design
SnakeAndLadder.cpp
int main() {
cout << "2. Medium (balanced)" << endl;
cout << "3. Hard (more snakes)" << endl;
int diffChoice;
cin >> diffChoice;
RandomBoardSetupStrategy::Difficulty diff;
switch(diffChoice) {
case 1: diff = RandomBoardSetupStrategy::EASY; break;
case 2: diff = RandomBoardSetupStrategy::MEDIUM; break;
case 3: diff = RandomBoardSetupStrategy::HARD; break;
default: diff = RandomBoardSetupStrategy::MEDIUM;
game = SnakeAndLadderGameFactory::createGame(choice, boardSize, diff);
board = new Board(boardSize);
// Custom game
int boardSize;
cout << "Enter board size (e.g., 10 for 10x10 board): ";
cin >> boardSize;
cout << "Choose custom setup type:" << endl;
Code for Snake and Ladder


Final UML Diagram
Dice
Player
Gamefactory
Snake
Ladder
Difficulty
EASY,
MEDIUM
HARD
BoardEntity
RandomBoardStrat
customCountStrat
setUpStrategy
setupBoard
(Board b);
Standard Strat
IObserver
updateCstring
msg);
ConsoleNotifier
Rule
Standard Rule

Lecture 34 : Build Snake and Ladder Game

Requirements

- → Size of the board should be scalable.
- → There can be game setup strategy like Random setup, custom set up, standard setup, etc.
- → Notifications (In-app)

UML ~~Design~~ Diagram Details

① We will be using top-down approach i.e. starting with main class then creating supportive class as needed.

② If for creating cells in our board we are thinking of 2D vector then it is no need as we can store it in 1D vector as no. will be consistent but issue of snake & ladder? → then map <snake, ladder> will be perfect. (details later)

③ So create orchestra class - Game and for playing game we will need Board then create Board class.

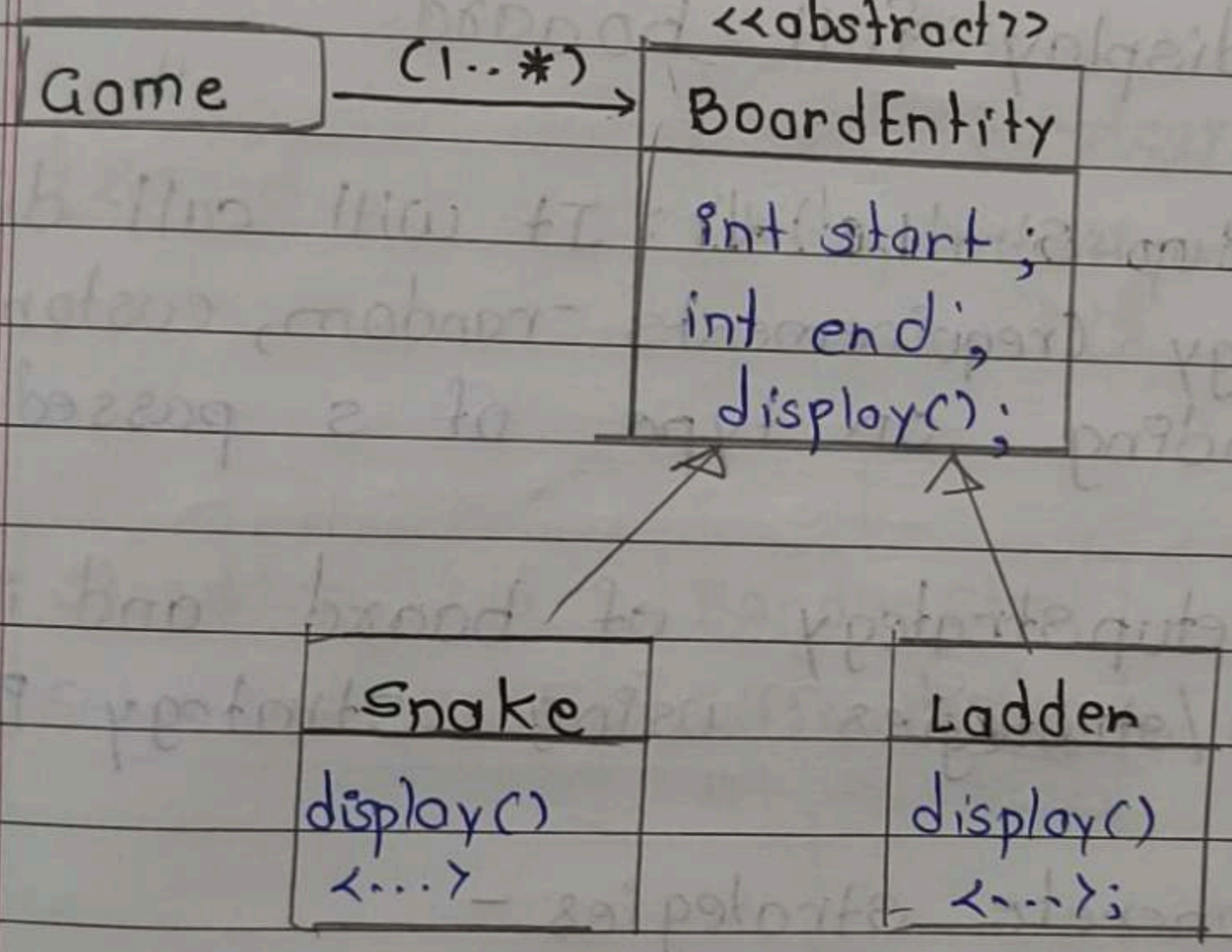
Game
Board board;

Board
int size
vector <Board Entity> en;

④ Now in Board class, first requirement is board scalability ∴ we will take size from client.

⑤ Now we have to display snake and ladders so we can use two separate vector to do so but instead we can use ~~some~~ Parent class - BoardEntity inherit and override by snake and ladder class

start end
→ snake move: Head —> tail
→ Ladder move : tail —> Head



⑥ Now in our board class we will use map as discussed - map <int, BoardEntity> mp;

index of
storing head if snake
storing index of tail if ladder

⑦ Why not using ID ? coz we have stored size and we know its continous value (1...size) so no need to give extra space.

⑧ Methods in Board class

① addEntity(BoardEntity) {} : Takes boardEntity cond adds in our vector and entity would know its start and end point.

② canAddEntity(pos) : We will check whether we can add entity at given pos. If there exist anything like Snake/Ladder then we can't add another & entity - Snake/Ladder.

③ display() : display full boaard

④ setupBoard(setup-Strat s) {} : It will call different type of Strategy (requirements - random, custom & standard) depending on type of s passed.

⑨ Let's create setupstrategy of board and its concrete classes/strategies using strategy Pattern

⑩ Let's discuss concrete strategies -

① Standard strat : It's the one which we play normally of (1-100) size and fixed place of snake and ladder.

② Random Strat : Here no. of snakes and ladders will depend on difficulty suppose if difficulty easy then 70% - ladder & 30% snake for hard → 30% ladder & 70% snake for medium → 50% - 50%

③ Custom strat : Here we will ask client how many no. of snake and ladder. Position of snake

and ladd
client Inp
use vec

• Method
① AddSn
② addL
③ setup

Board

Standard
SetupBoa

⑪ Now,
be sca
no. on
and ro

Final UML Diagram

Game
- Board board;
- Dice dice;
- dequeue<Players> player;
- Rules rules;
- vector<IObserver> o;
- bool gameOver;
- addPlayer(Player P)<-->;
- addObserver(IObserver o)<-->;
- notify(String msg)<-->;
- play()<-->

Board
- int size;
- vector<BoardEntity> sl;
- map<int, BoardEntity> entities;
- canAddEntity(int pos)<-->
- addBoardEntity(BoardEntity * e)<-->;
- setupBoard(setupstrats)<>;
- display()<-->;

(1..*)

<<abstract>> IObserver
- updateString msg);

ConsoleNotifier
- update(String msg)<-->;

<<abstract>> Rule
- isValidMove(pos, diceV, bs);
- calcNewPos(pos, diceV, Board b);
- checkWin(pos, b);

StandardRule
- isValidMove(p, dv, bs)<-->;
- calcNewPos(p, dv, b)<-->;
- checkWin(p, bs)<-->;

<<abstract> BoardEntity
- int start;
- int end;
- display();

setUpStrategy
- setupBoard(Board b);

GameFactory
- createGame()<-->;

Snake
- display()<-->

Ladder
- display()<-->;

RandomBoardStrat
- Difficulty difficulty;
- setupBoard(Board b)<-->;

StandardStrat
- setupBoard(Board b)<-->;

Dice
- int faces;
- roll()<-->;

Player
- int id;
- string name;
- int pos;
- int score;

Difficulty
- EASY,
- MEDIUM,
- HARD

customcountStrat
- int snake, ladders;
- bool randomPos;
- vector<Pair<>> snakePos;
- vector<Pair<>> ladderPos;
- addSnakePos(s,e)<-->;
- addLadderPos(s,e)<-->;
- setupBoard(Board b)<-->;

(15) Now last step is Notification for it we will create same hierarchy as before. IObserver and then consoleNotifier.

(16) We will add more methods to our Game class -
① addObserver(ob)
② addPlayer(p)
③ bool isGameOver; → checkWin in Rule class
④ play(); ⟶ while(! isGameOver){
⑤ notify() ≡
}

(17) And finally our Factory class - GameFactory having method createGame()

Flow of Game -

① Client calls play() → then we will find currentPlayer then roll dice after that checkValidMove → if yes then check for calcNewPos as to check for snake & ladders (while(!.GameOver)<

② To check it, it wil go to map + entities in our board class and depending if it is there or not newPos will be added.

③ Then we will checkWin() condition if true then true and GameOver → true & loop terminates

④ If not then again find new player through dequeue ~~and~~ and repeat step till someone wins

(12) Now will create dequeue in Game class for Players and then create Player class

Player
int id;
string name;
int pos; → current position of player
int score; → no. of wins of a player
// G & S

Game —(1..*)→ Player

(13) Now in main Game class we will add Rules object as Board is our dumb object if following SRP as responsibility of boadd is to manage it and no check rules satisfies or not.

(14) ∴ Creating Rule class - strategy Pattern,
Right now we are only using standardRule but if incase we need different Rules so it should be added easily

Rule	
isValidMove(pos, dvalue, board);	→ checks valid move
checkWin(pos, bs); → board size	if we are at gg, dice roll and its 2 then for winning we need 1 so we can't move 2 so invalid in that case
calcNewpos(pos, dvalue, board);	

Game → Rule

if pos == size
then win

if pos == size
then pos += dvalue
or checks for ladder & snake
if yes then its condition

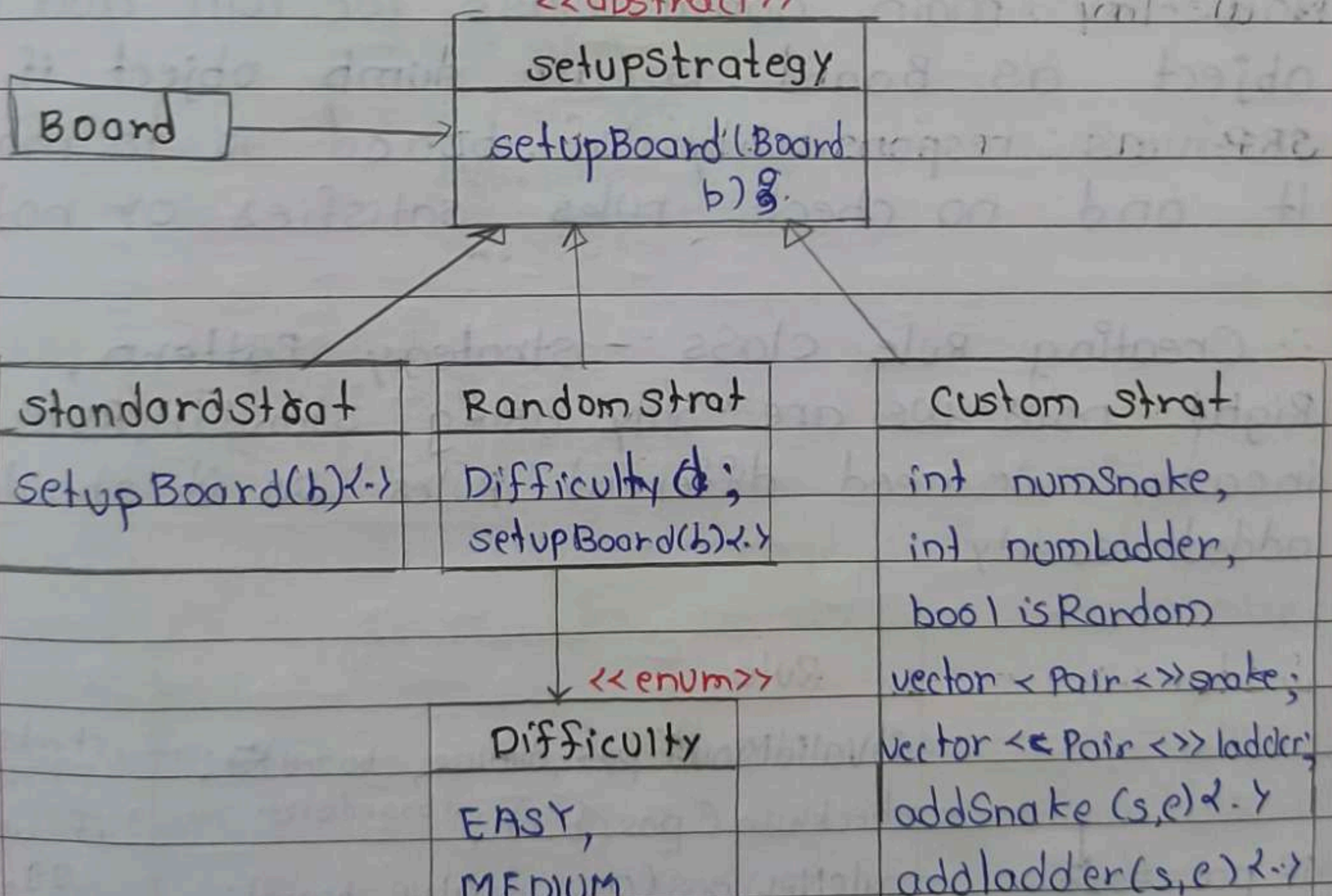
StandardRule	
isValidMove(pos, dv, bs)<->	
checkWin(p, bs)<->	
calcNewpos(p, dv, b)<->	

and ladder should be random (by our side) or clientInput - if client giving then we will use vector <pair <int, int> > snake & same for ladden

- int → start pt of snake/ladder
- int → end pt of snake/ladder

- Methods - (start, end)
  1. addSnake (s, e) {..}
  2. addladder (s, e) {..}
  3. setupBoard (b) {..};

} Add snakes and ladders in our board



(11) Now, we will create Dice class. it should be scalable :. int faces → will show maximum no. on dice
and roll() method.

Dice
int face;
roll(){..};