30 April 2024 20:40

LOW LEVEL DESIGN (LLD)

① SOLID Principle

S → Single Responsibility
O → Open for extension | closed for modification
L → Liskov Substitution Principle
I → Interface Segmented Principle
D → Dependency Injection Principle

Advantage
① Avoid duplicate code
② Easy to maintain
③ Easy to understand
④ Flexible software
⑤ Reduce complexity.

[Single Responsibility Principle]

A class should have only reason to change

Maker class

```
class Maker {
    String name;
    String color;
    int year;
    int price;
// All arg constructor
//Getter & setter
}
```

Invoice class

```
class Invoice {
    Maker maker;
    private int quantity;
public int calculateTotal() {
    int price = (maker.price) * quantity;
    return price;
}
public void printInvoice() { }
public void saveToDB() { }
```

* there are multiple reason for the Invoice class to change
← not following Single Responsibility.

what we can do.

change the class into 3 different class

```
class InvoiceCalculation {
    Maker maker;
    private int qty;
    public calculateTotal() {
        return maker.price * qty;
    }
}
```

```
class InvoicePrint {
    InvoiceCalculation invoice;
    //parametrized constructor
    public void print() {
        ---
    }
}
```

```
class InvoiceSaveToDB {
    InvoiceCalculation invoice;
    //parametrize constructor
    public void print() {
        ---
    }
}
```

* here each class has only one reason to change.

Open/close Principle :→

Open for Extension closed for modification

Don't touch already tested class. For example: Let say we want to save in file system as well.
so instead of modifying existing [InvoiceSaveToDB class], i will do below. :→

```
interface InvoiceDao {
    public void save(Invoice invoice);
}
```

```
interface InvoiceDao {
    public void save (Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {
    @Override
    public void save (Invoice invoice) {
        //save to DB
    }
}

class FileInvoiceDao implements InvoiceDao {
    @Override
    public void save (Invoice invoice) {
        //save to file
    }
}
```

Liskov Substitution Principle :→

If class B is subclass of class A, then we should be able to replace object of A with B w/o breaking the behavior of the program.

* Subclass should extend the capabilities of parent class not narrow it down.

```
interface Bike {
    void turnOnEngine ();
    void accelerate ();
}

class Motorcycle implements Bike {
    boolean isEngineOn;
    int speed;
    public void turnOnEngine () {
        isEngineOn = true;
    }
}
```

here bicycle does not have engine, so it can't perform any action & it is restricting the Bike interface

```
class Bicycle implements Bike {
    int speed;
    public void turnOnEngine () {
        throw new AssertionError ("...");
    }
}
```

→ changing the behavior

Interface Separation Principle :

Interfaces should be such, that client should not implement fn's that they do not need.

```
interface RestaurantEmployee {
    void washDishes ();
    void serveCustomers ();
    void cookFood ();
}
```

→ what if we have

Waiter class — it can only serve dishes
Helper class — it can only wash dishes
Chef class — it can only cook dishes

* so we are seeing the interface is not proper as unnecessary fn's will be given to the different classes.

Instead do this;

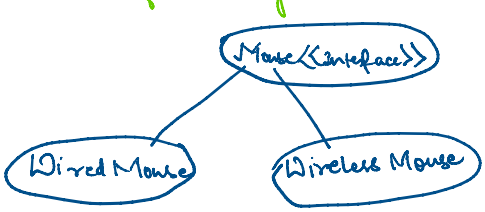
```
interface WaiterInterface {
    void serveCustomer ();
    void takeOrder ();
}

interface ChefInterface {
    void decideMenu ();
    void cookFood ();
}
```

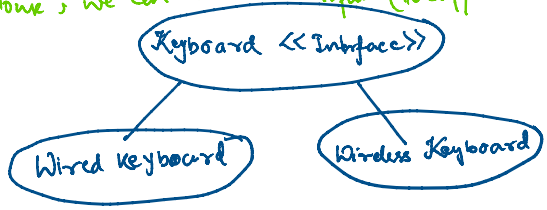
this makes more sense as each class will only have necessary fn's

Dependency Injection

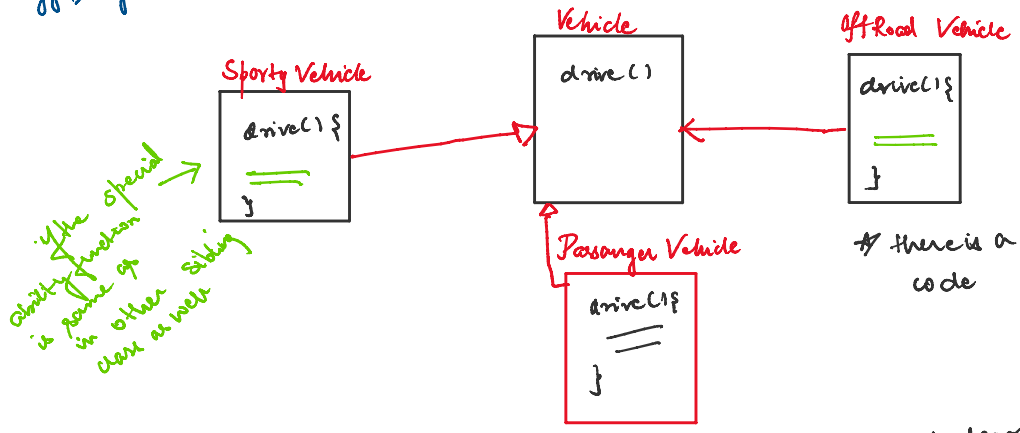
Class should depend on interfaces rather than concrete class.

Instead of sending Wired Keyboard & Wireless Mouse, we can create interface (to support all other types)

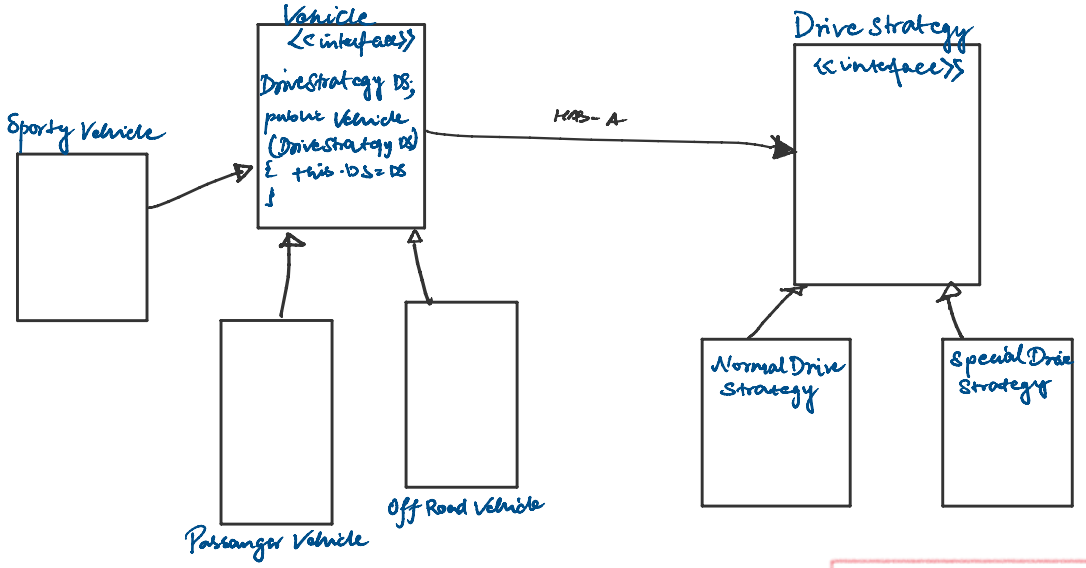




① Strategy Design Pattern :→



Lets create an interface "DriveStrategy" which has "NormalDriveStrategy" & "SpecialDriveStrategy" class (sub class). Now child has the permission to decide strategy. Using constructor Injection in Vehicle interface which has A DriveStrategy object.



Code :→

```
public class Vehicle {
DriveStrategy ds;
public Vehicle (DriveStrategy dS){
 this.dS=dS
}
public void drive() {
   ds.drive();
}}
```

here if you see, the code is getting duplicated & this can be avoided by using strategy design pattern

```
public class SportsVehicle{
   public void drive(){
     Sysout("sports drive capability");
   }}
```

```
public class OffRoadVehicle{
   public void drive(){
     SysOut("sports drive capability");
   }}
```

```
public class DriveStrategy {
   public void drive();
}
```

```
public class NormalDriveStrategy
   implements DriveStrategy
{
   public void drive(){
     Sysout("Normal Drive Strategy");
   }}
```

```
public class SportsDriveStrategy
   implements DriveStrategy
{
   public void drive(){
     Sysout("Sports Drive Capability");
   }
```

```
public class OffRoadVehicle extends Vehicle{
    OffRoadVehicle(){
        super(new SportsDriveStrategy());
    }
}
```

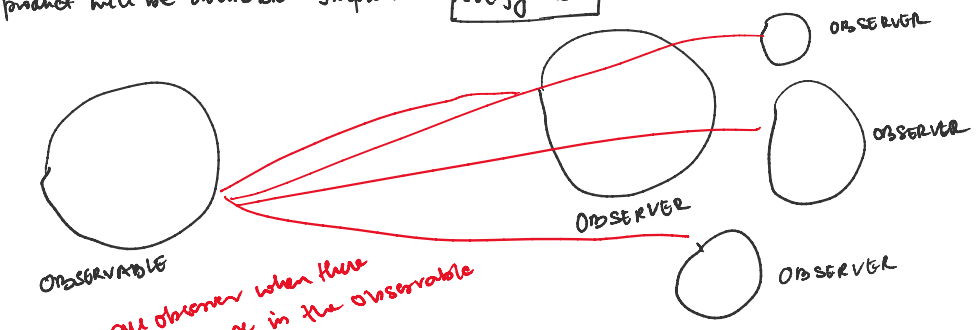
→ here child class is responsible for changing the strategy & this is with help of "Has A" relation & constructor injection

Q: When to use Strategy design pattern?
Ans:→ when child class have similar code which is not present in the parent then go for StrategyDesignPattern.

---

2) OBSERVER Design Pattern :→

Ques:– Suppose you go on Amazon, and you find your product, out of Stock. There is a button to "notify Me" when the product will be available. Implement Notify Me



update all observer when there is state change in the observable

OBSERVABLE <<interface>>
List<Observer> obList
– add(Observer obj);
– remove(Observer obj);
– notify()
– setData()
– getData()

(1 - *)
has-a

OBSERVER <<interface>>
update()

↳ then create concrete class for both the interface

ObserverImpl <<concrete class>>

```
Observable observable;
public ObserverImpl(Observable observable){
    this.observable = observable;
}
public void update(){
    observable.getData();
}
```

ObservableImpl <<Concrete class>>

```
int data;
List<Observer> obList;
public void add(Observer obj){
    obList.add(obj);
}
public void remove(Observer obj){
    obList.remove(obj);
}
public void notify(){
    for(observer ob : obList){
        ob.update();
    }
}
public void setData(int val){
    data = val;
    notify();
}
```

↳ instead of using instance of in update we are using constructor injection

---

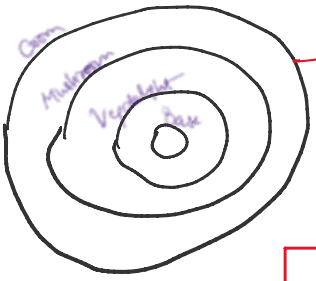
3) Decorator Pattern :→ If you take the example of Base pizza, it has MargheritaPizza, VegDelight, ChickenDelight & also it has toppings as ExtraCheese, Jalpeno, ExtraCorn, ExtraSpices

So, if we are not using the decorator design pattern, we need to create class for each of the permutations & combination, which will result in class explosion.

→ Only pattern which has A and ISA relation

So, if we are not using the decorator design pattern, we need to create class for each of the permutations & combination, which will result in class explosion.

→ Only pattern which HAS A and IS A relation



→ here we can add any number of decorators & still the class won't explode.

```
public abstract class BasePizza {
    public int cost();
}
```

is a

```
public abstract class ToppingDecorator extends BasePizza {
}
```

has a

```
public class VegDelight extends BasePizza {
    @Override
    public int cost() {
        return 100;
    }
}
```

is a

```
public class ChickenDelight extends BasePizza {
    @Override
    public int cost() {
        return 150;
    }
}
```

is a

```
public class Mushroom extends ToppingDecorator {
    BasePizza basePizza;
    @Override
    public int cost() {
        return basePizza.cost() + 10;
    }
}
```

has a

```
public class ExtraCheese extends ToppingDecorator {
    BasePizza basePizza;
    @Override
    public int cost() {
        return basePizza.cost() + 20;
    }
}
```

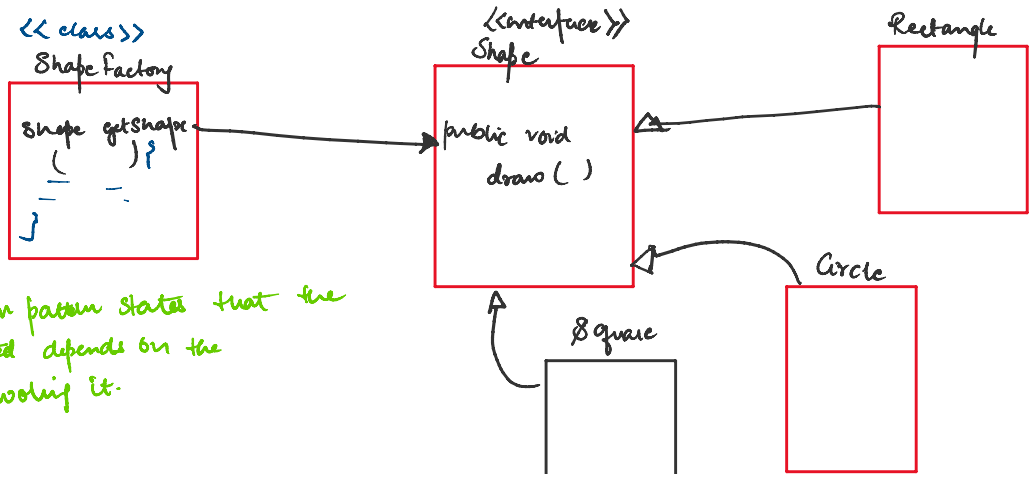
Client class

```
psvm (String[] args) {
    BasePizza = new ExtraCheese(new Mushroom(new VegDelight()));
    basePizza.cost();
}
```

→ here if you see, how the no. of classes have reduced & still we are able to add toppings on the pizza to get the result.

---

⑭ Factory Vs Abstract Factory Design pattern :→



Factory design pattern states that the object created depends on the subclasses involving it.

subclass implementing it.

```
public class ShapeFactory {
    Shape getShape (String input){
      Switch (input){
      case "CIRCLE":
            return new Circle();
      case "RECTANGLE":
            return new Rectangle();
      default:
            return null;
    } }
```

```
public interface Shape {
      public void draw();
}
```

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        SysOut ("Rectangle");
} }
```

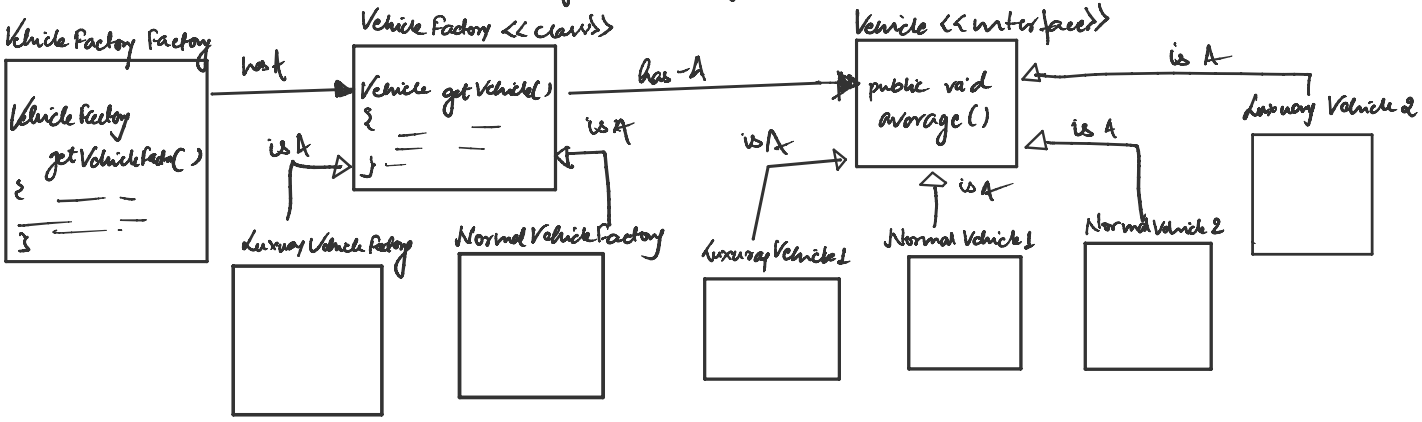
```
public class Circle implements Shape {
    @Override
    public void draw() {
        SysOut ("Circle");
} }
```

Main.java

```
{ psvm (String [] args) {
    ShapeFactory sf = new ShapeFactory();
    Shape shapeObj1 = sf.getShape ("RECTANGLE");
     shapeObj1.draw(); ——→ this gives Rectangle
    Shape circle = sf.getShape ("CIRCLE");
     circle.draw(); ——→ this gives Circle
} }
```

---

⑤ Abstract Factory Pattern :→ It is factory of factory design pattern




```
public class VehicleFactoryFactory {
    VehicleFactory getVehicleFactory
        (String input) {
    switch (input){
    case "LUXURY":
          return new Luxury
                VehicleFactory();
    case "NORMAL":
          return new NormalVehicle
```

```
public interface VehicleFactory{
    Vehicle getVehicle (String input);
```

```
public class LuxuryVehicleFactory
    implements VehicleFactory{
@Override
Vehicle getVehicle (String input){
    switch (input){
```

```
public interface
    Vehicle {
    public void drive();
```

```
public class LuxuryVehicle1 implements
    Vehicle {
@Override
public void drive(){
    SysOut ("Luxury1 drive");
```

```
case "NORMAL":
    return new NormalVehicle
                Factory();
default: return null;
}
}
```

```
switch (input){
case "BMW" :
    return new LuxuryVehicle1();
case "AUDI" :
    return new LuxuryVehicle2();
default :
    return null;
}
}
```

```
@Override
public void drive() {
    SysOut ("Luxury1 drive");
}
```

```
public class LuxuryVehicle2 implements
  Vehicle {
  @Override
  public void drive() {
    SysOut ("Luxury2 drive");
  } }
```

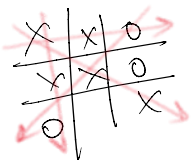
```
public class NormalFactory
    implements VehicleFactory {
  @Override
  public Vehicle getVehicle(String
                    input)
  {
    switch(input) {
    case: "HYUNDAI":
        return new NormalVehicle1();
    case : 'ALTO':
        return new NormalVehicle2();
    default: return null;
  } }
}
```

```
public class NormalVehicle1 implements
  Vehicle {
  @Override
  public void drive() {
    SysOut ("Normal 1 drive");
  } }
```

```
public class NormalVehicle2 implements
  Vehicle {
    @Override
    public void drive() {
      SysOut("Normal 2 drive");
  } }
```

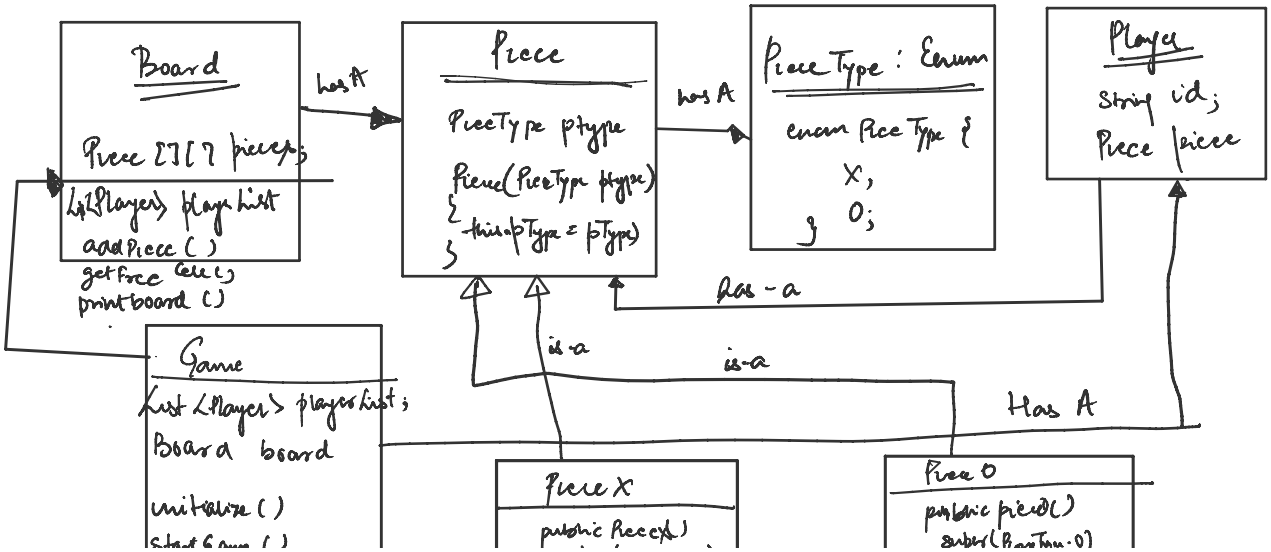
```
Main.java
{ psvm (String [] args) {
    VehicleFactoryFactory vfactoryfactory = new VehicleFactoryFactory();
    VehicleFactory vfactory = vfactoryfactory.get("LUXUARY").get("BMW");
    vfactory.drive();  ⟶ this prints Luxuary1 drive
} }
```

Example Question : Design LLD for Tic Tac Toe



for designing Tic Tac Toe, we should first know the objects needed :

Object needed
- PieceType
- Piece
- Board
- Player



```
Board
initialize()
startGame()
```

```
PieceX
public PieceX()
  super(PieceType.X)
```

```
PieceO
public PieceO()
  super(PieceType.O)
```

```
public enum PieceType {
    X,
    O;
}
```

```
public class PieceX extends Piece {
    public PieceX() {
        super(PieceType.X);
    }
}
```

```
public class PieceO extends Piece {
    public PieceO() {
        super(PieceType.O);
    }
}
```

```
public class Piece {
    PieceType ptype;
    public Piece(PieceType ptype) {
        this.ptype = ptype;
    }
}
```

```
public class Player {
    Piece piece;
    String id;
    public Player(Piece piece, String id) {
        this.piece = piece;
        this.id = id;
    }
}
```

missing method

```
public void printBoard() {
    for (i -> 0 to size)
        for (j -> 0 to size)
            if (board[i][j] != null)
                System.out.print(board[i][j].pType.name() + " ");
            else
                System.out.print(" ");
        System.out.println();
    }
```

Board.java

```
public class Board {
    public int size;
    public Piece[][] board;
    public Board(int size) {
        this.size = size;
        board = new Piece[size][size];
    }
    public boolean addPiece(int row, int col, Piece piece)
    {
        if (board[row][col] != null)
            return false;
        board[row][col] = piece;
    }   return true;
    public List<Pair<Integer, Integer>> getFreeCells() {
        List<Pair<Integer, Integer>> freeCells = new ArrayList<>();
        for (int i=0; i<size; i++) {
            for (int j=0; j<size; j++) {
                if (board[i][j] == null){
                    Pair<Integer, Integer> rowCol = new
                                                Pair<>(i,j);
                    freeCells.add(rowCol);
                }
            }
        }
    }   return freeCells;
}
```

Game.java

```
public class TicToeGame {
    Queue<Player> players;
    Board gameBoard;
    TicToeGame() {
        initializeGame();
    }
    public void initializeGame() {
        players = new LinkedList<>();
        PieceX cross = new PieceX();
        Player p1 = new Player("p1", cross);
        PieceO naught = new PieceO();
        Player p2 = new Player("p2", naught);
        players.add(p1);
        players.add(p2);
        gameBoard = new Board(3);
    }
    public String startGame() {
        boolean isWinner = true;
        while (isWinner){
            Player pTurn = players.removeFirst();
            gameBoard.printBoard();
            List<Pair<Integer, Integer>> freeSpace =
                gameBoard.getFreeCells();
            if(freeSpace.isEmpty()){
                isWinner = false;
            }   continue;
            System.out.println("Player: " + pTurn.getId() + "Enter row & col");
            Scanner sc = new Scanner(System.in);
            String s = sc.nextLine();
```

```
public boolean isThereWinner(int row, int col, PieceType pType) {
    boolean rowMatch = true;
    boolean colMatch = true;
    boolean diagonalMatch = true;
    boolean antiDiagonalMatch = true;
    for(int i=0; i<size; i++) {
```

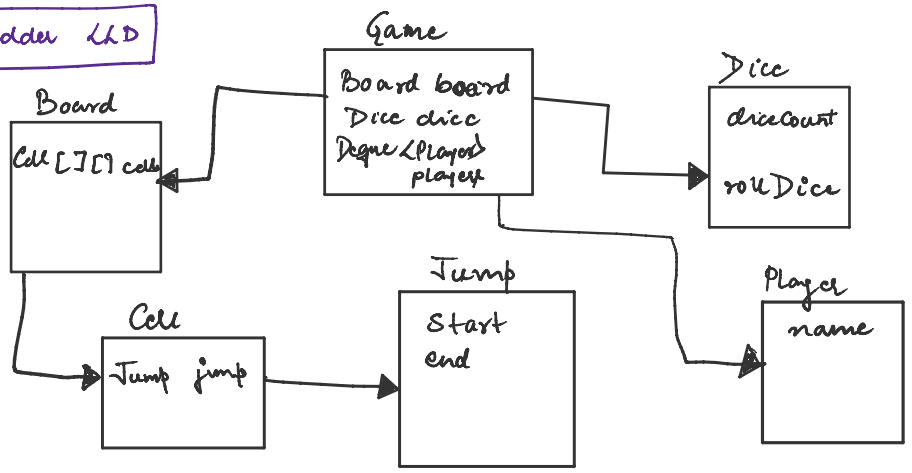
```
boolean diagonalMatch = true;
boolean antiDiagonalMatch = true;
for(int i=0; i<gameBoard.size; i++) {
    if(gameBoard[row][i] == null || gameBoard[row][i].pType != pType)
        rowMatch = false;
    if (gameBoard[i][col] == null || gameBoard[i][col].pType != pType)
        colMatch = false;
}
for (int i=0, j=0; i<gameBoard.size; i++, j++) {
    if(gameBoard.board[i][j] == null || gameBoard[i][j].pType != pType)
        diagonalMatch = false;
}
for(int i=0, j=gameBoard.size-1; i<gameBoard.size; i++, j--)
{
    if(gameBoard.board[i][j] == null || gameBoard.board[i][j].pType != pType)
        antiDiagonalMatch = false;
}
return rowMatch || colMatch || diagonalMatch || antiDiagonalMatch;
}
```

```
System.out.("Player: "+pTurn.getId()+" Enter row & col");
Scanner sc = new Scanner(System.in);
String s = sc.nextLine();
String[] values = s.split(" ");
int inputRow = Integer.valueOf(values[0]);
int inputCol = Integer.valueOf(values[1]);
boolean pieceAdd = gameBoard.addPiece(
        inputRow, inputCol, pTurn.piece());
if(!pieceAdd){
System.out.println("Incorrect position! Try again!!");
players.addFirst(pTurn);
continue;
}
players.addLast(pTurn);
boolean winner = isThereWinner(inputRow, inputCol,
        pTurn.getPiece().getPieceType());
if (winner)
    return pTurn.getId();
}
return "tie";
```

Example Question : Snake & Ladder LLD

Object needed

- Dice
- Player
- Board
- Cell
- Snake & ladder (jump)



Dice.java

```
public class Dice {
    int diceCount;
    int min = 1;
    int max = 6;
    public int rollDice() {
        int diceNow = 0;
        int total = 0;
        while ( diceNow < diceCount)
        {
            total += ThreadLocalRandom.current()
                    .nextInt(min, max+1);
        diceNow++;
    } return total;
}
```

Cell.java

```
public class Cell {
    Jump jump;
}
```

Player.java

```
public class Player {
    String id;
    int currPos;
    public Player(String id,
        int currPos) {
    this.id = id;
    this.currPos = currPos;
}}
```

Jump.java

```
public class Jump {
    int start;
    int end;
```

Board.java

```
public class Board {
    Cell[][] cells;
    Board(int size, int snakes, int ladders) {
        initializeCells(size);
        addSnakesLadders(cells, snakes, ladders);
```

Game.java

Game.java

```
public class Game {
    Board board;
    Dice dice;
    Player winner;
    Deque<Player> players;

    public Game() {
        initializeGame();
    }
    public void initializeGame() {
        board = new Board(10,5,4);
        dice = new Dice(1);
        winner = null;
        addPlayer();
    }
    public void addPlayer() {
        Player p1 = new Player("p1", 0);
        Player p2 = new Player("p2", 0);
        players.add(p1);
        players.add(p2);
    }
    public void startGame() {
        while(winner == null){
            Player pTurn = findPlayerTurn();
            SysOut("player:" + pTurn.id + "position" +
                    pTurn.currentPos);
            //roll Dice
            int diceNumber = dice.rollDice();
            int playerNewPos = pTurn.currentPost diceNumber;
            playerNewPos = jumpCheck(playerNewPos);
            pTurn.currentPos = playerNewPos;
            SysOut("player" + pTurn.id + "cur position" +
                    playerNewPos);
            //Check winning strategy
            if(playerNewPos >= board.cells.length * board.cells.
                    length - 1) {

                winner = pTurn;
            }
            SysOut("Winner is:" + winner.id);
        }
    private Player findPlayerTurn() {
        Player pTurn = players.removeFirst();
        players.addLast(pTurn);
    } return pTurn;
    private int jumpCheck(int playerPos) {
        if (playerPos > board.cell.length * board.cell length-1)
                return playerPos;
        cell cell = board.getCell(playerPos);
        if (cell.jump != null && cell.jump.start ==
```

```
    Board(int size, int snakes, ... ) {
        initializeCells(size);
        addSnakes&Ladders(cells, snakes, ladders);
    }
    private void initializeCells(int boardSize){
        cells = new Cell[boardSize][boardSize];
        for(int i=0; i< boardSize; i++){
            for(int j -> 0 to boardsize)
                Cell obj = new Cell();
        } } cells[i][j] = obj;
    }
    private void addSnakes&Ladders(Cell[][] cells, int snakes,
                                   int ladders) {
        while(snakes > 0) {
            int snakeHead = ThreadLocalRandom.current().nextInt(
                    1, cells.length * cells.length -1);
            int snakeTail = ThreadLocalRandom.current().nextInt(
                    1, cells.length * cells.length -1);
            if (snakeTail >= snakeHead)
                continue;
            Jump snakeObj = new Jump();
            snakeObj.start = snakeHead;
            snakeObj.end = snakeTail;
            Cell obj = getCell(snakeHead);
            obj.jump = snakeObj;
        } snakes--;
        while(ladders > 0){
            int ladderHead = ThreadLocalRandom.current().nextInt(1,
                    cells.length * cells.length -1);
            int laddaTail = ThreadLocalRandom.current().nextInt(1,
                    cells.length * cells.length -1);
            if (ladderHead >= ladderTail)
                continue;
            Jump ladderObj = new Jump();
            ladderObj.start = ladderHead;
            ladderObj.end = ladderTail;
            Cell obj = new Cell();
            obj.jump = ladderObj;
        } ladders--;
    }
    Cell getCell(int playerPos) {
        int boardRow = playerPos / cells.length;
        int boardCol = playerPos % cells.length;
    } return cells[boardRow][boardCol];
}
```

```
return [-1, -1];
cell cell = board.getCell(playerPos);
if (cell.jump != null && cell.jump.start == playerPos)
    String jumpBy = (cell.jump.start < cell.jump.end) ? "ladder" : "snake";
    System.out("Jump done by" + jumpBy);
    return cell.jump.end;
return playerPos;
```

Ques :→ PARKING SPOT (Low Level Design)

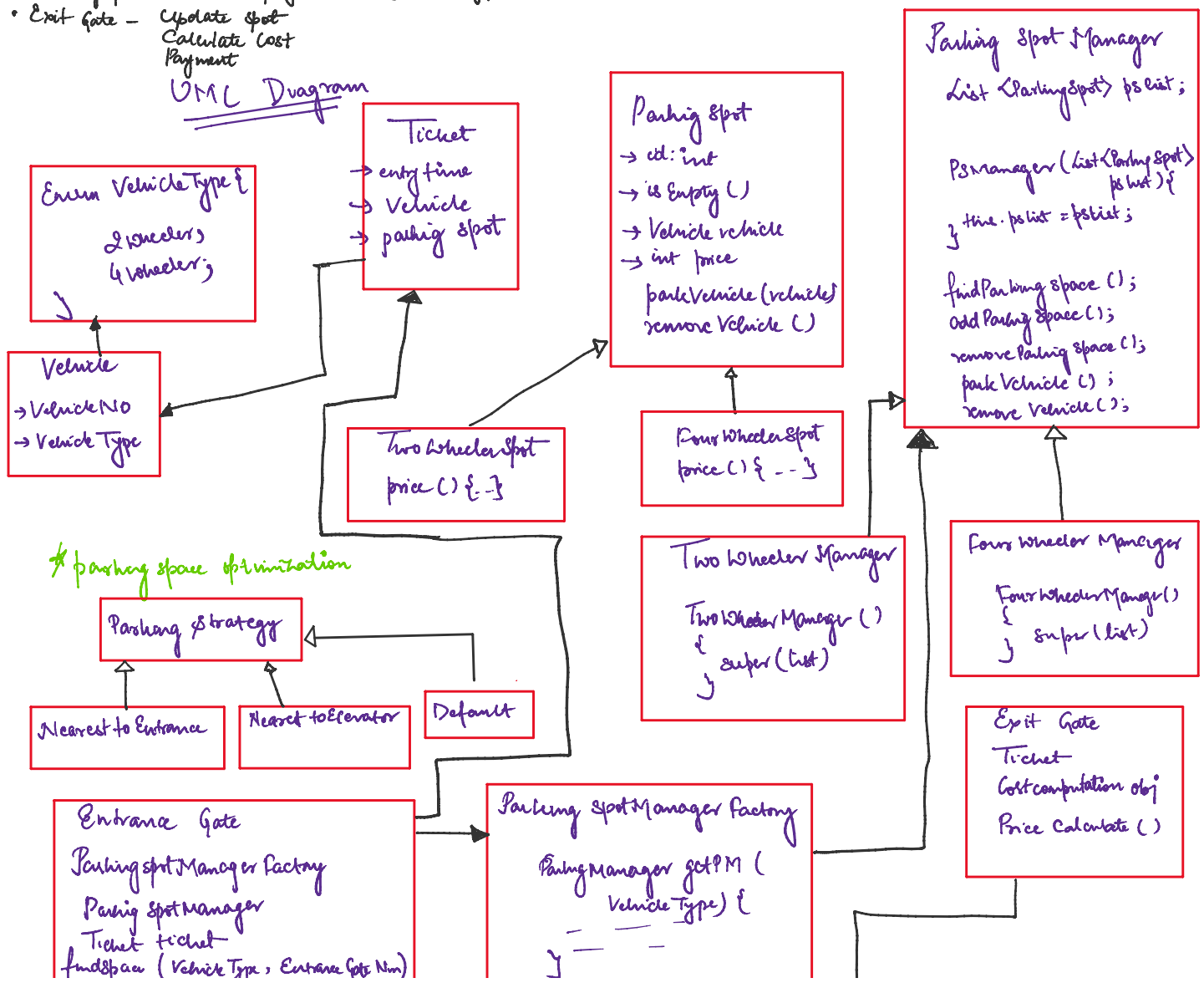
Clarification On Requirement

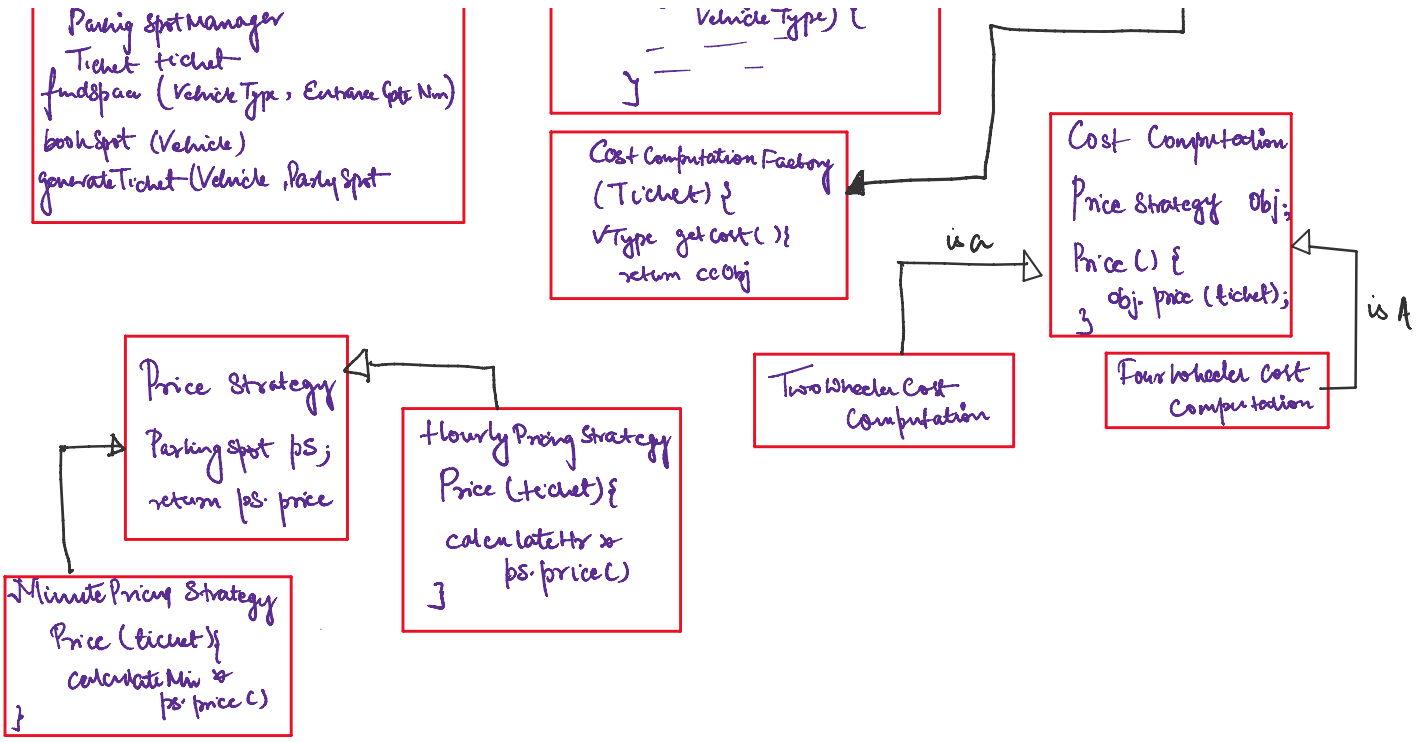
Q :- How many vehicle type?
Q :- How many floors ?
Q :- How many entrance/exit gate
Q :- Strategy for parking vehicle

Object

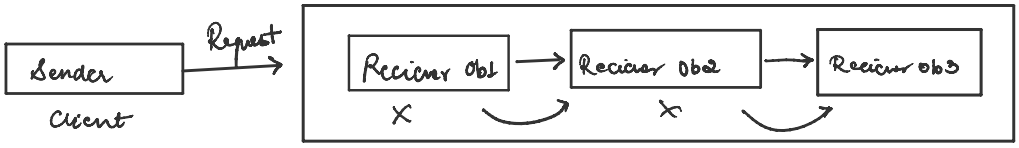
- Vehicle — Vehicle No, Vehicle Type → enum { 2 Wheeler, 4 Wheeler }
- Ticket — Entry time, parking spot
- Entrance Gate — findSpace, update spot, generate Ticket
- Parking Spot — id, isEmpty(), Vehicle, Price, Type
- Exit Gate — Update Spot, Calculate Cost, Payment

UML Diagram





Chain Of Responsibility :→ * Design Logger




```
public abstract class LogProcessor {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;
    LogProcessor nextLoggerProcessor;
    LogProcessor (LogProcessor loggerProcessor) {
        this.nextLoggerProcessor = loggerProcessor;
    }
    public void log (int logLevel, String message) {
        if (nextLoggerProcessor != null)
            nextLoggerProcessor.log (logLevel, message);
    }
}
```

```
public class InfoLogProcessor extends
        LogProcessor {
    InfoLogProcessor (LogProcessor nextLogProcessor)
    {
        super (nextLogProcessor);
    }
    public void log (int logLevel, String message)
    {
        if (logLevel == INFO)
            SysOut ("INFO:" + message);
        else
            super.log (logLevel, message);
    }
}
```

```
public class DebugLogProcessor extends LogProcessor {
    DebugLogProcessor (LogProcessor nextLogProcessor) {
        super (nextLogProcessor);
    }
    public void log (int logLevel, String message) {
        if (logLevel == DEBUG)
            SysOut ("DEBUG:" + message);
        else
            super.log (logLevel, message);
    }
}
```

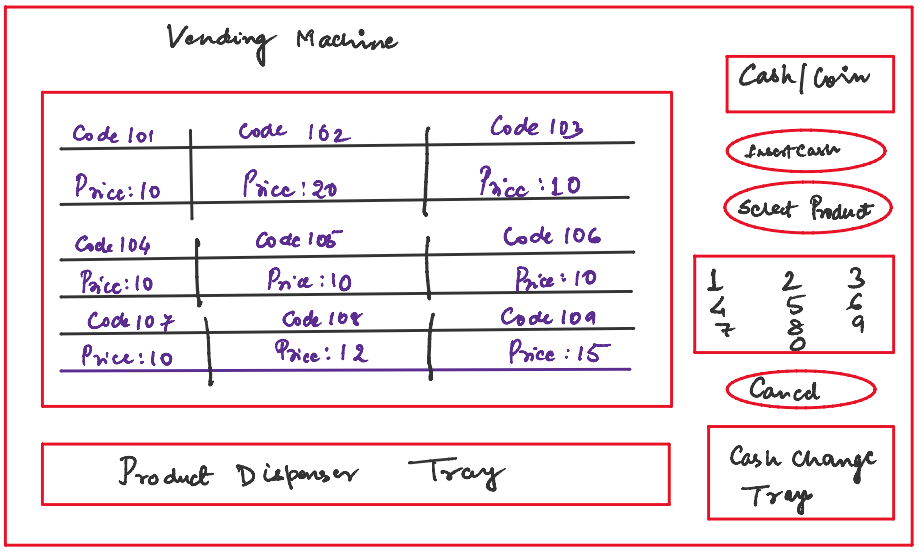
```
public class ErrorLogProcessor extends
        LogProcessor {
    ErrorLogProcessor (LogProcessor nextLogProcessor)
    {
        super (nextLogProcessor);
    }
    public void log (int logLevel, String message) {
        if (logLevel == ERROR)
            SysOut ("ERROR:" + message);
        else
            super.log (logLevel, message);
    }
}
```

```
            SysOut ("DEBUG:" + message);
        else
            super.log(logLevel, message);
    }
}
```

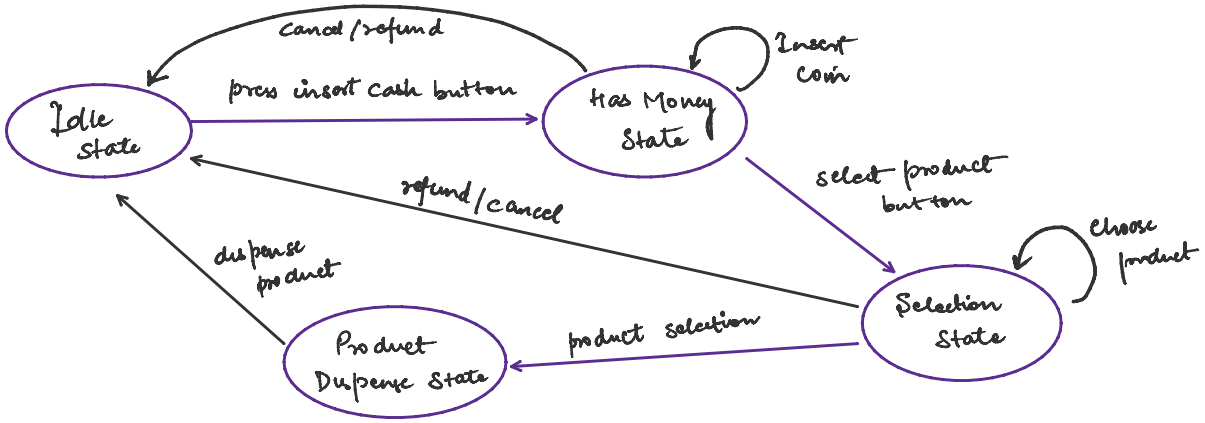
```
public class Main {
    public static void main (String[] args) {
        LogProcessor logObject = new InfoLogProcessor (new DebugLogProcessor (new ErrorLogProcessor (null)));
        logObject.log (LogProcessor.ERROR, "exception happened");
        logObject.log (LogProcessor.DEBUG, "need to debug this");
        logObject.log (LogProcessor.INFO, "just for info");
    }
}
```

Ques: Design Vending Machine (LLD)

It's a State Design pattern → State design pattern says, it will implement only those fns which are are necessary for that particular state, also once the state is completed, it will move to the next step.



* Happy flow



① Idle	→ press insert cash button
② hasMoney	→ insert coin → select product button ↩ cancel / refund
③ Selection	→ choose product → cancel / refund → return change
④ Dispensing	→ product dispense

Ques:- HashMap Internal Implementation

How to find the table size on given capacity.

// Returns a power of 2 size for the given target capacity.

```
static final int tableSizeFor (int capacity) {
    int n = capacity-1;
    n |= n>>>1;
    n |= n>>>2;
    n |= n>>>4;
    n |= n>>>8;
    n |= n>>>16;
    return (n<0) ? 1 : (n>=MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n+1;
}
```

* Approach to reach on above logic:

'>>>' - Similar to >> (right shift) but this considers unsigned bit as well.

So, if +ve integer is there, both works same

in case of -ve integer, ">>>" this is only effective.

now, why "capacity-1" ?

So, let's say we have capacity of 15 ——— So closest to 15 would be 2^4 → (16)

Binary for 15 —

16 8 4 2 1