Pre-requisite

- OOP's Concepts
- SOLID Principles

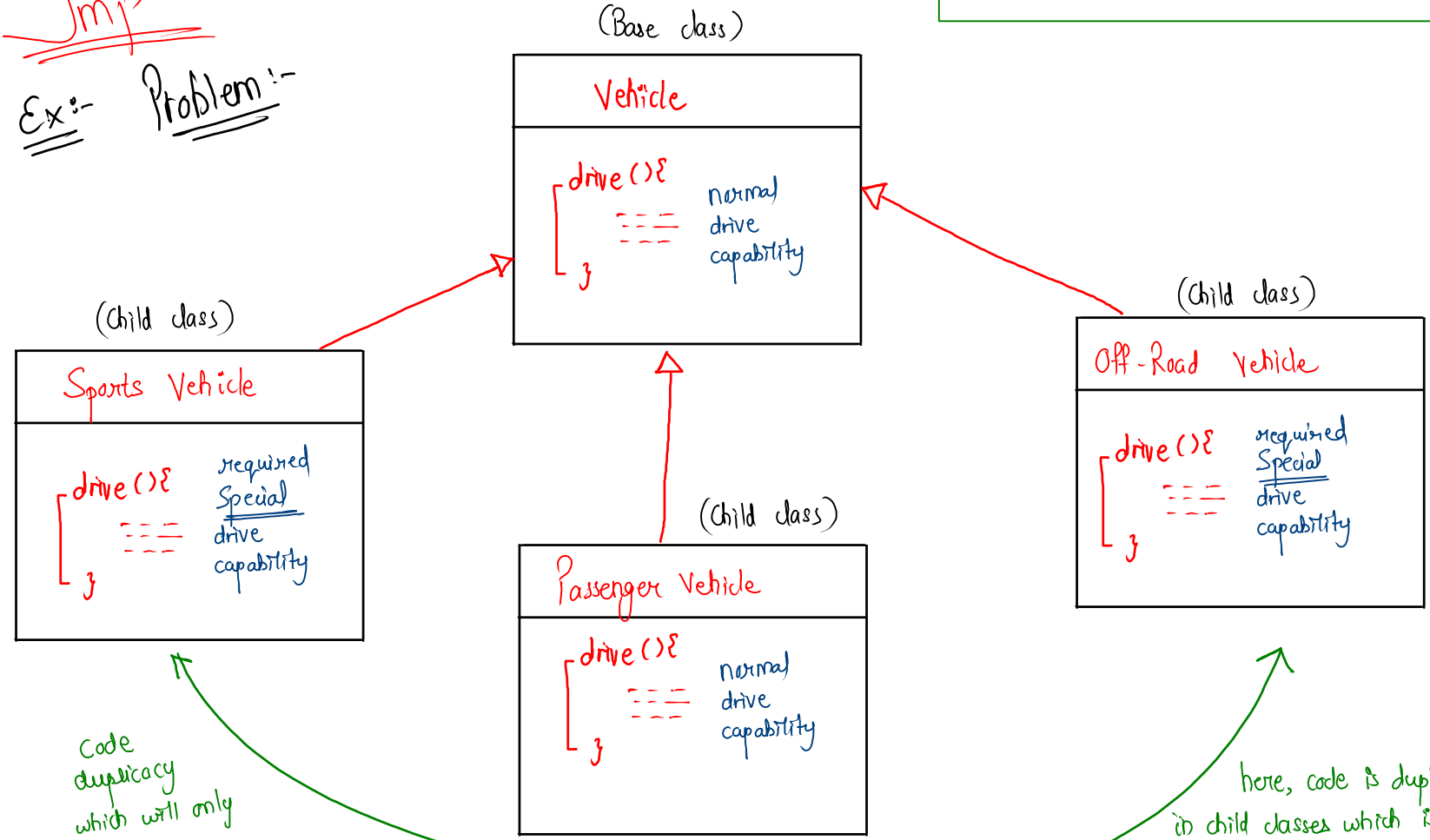
⇒ Strategy Design Pattern

Imp

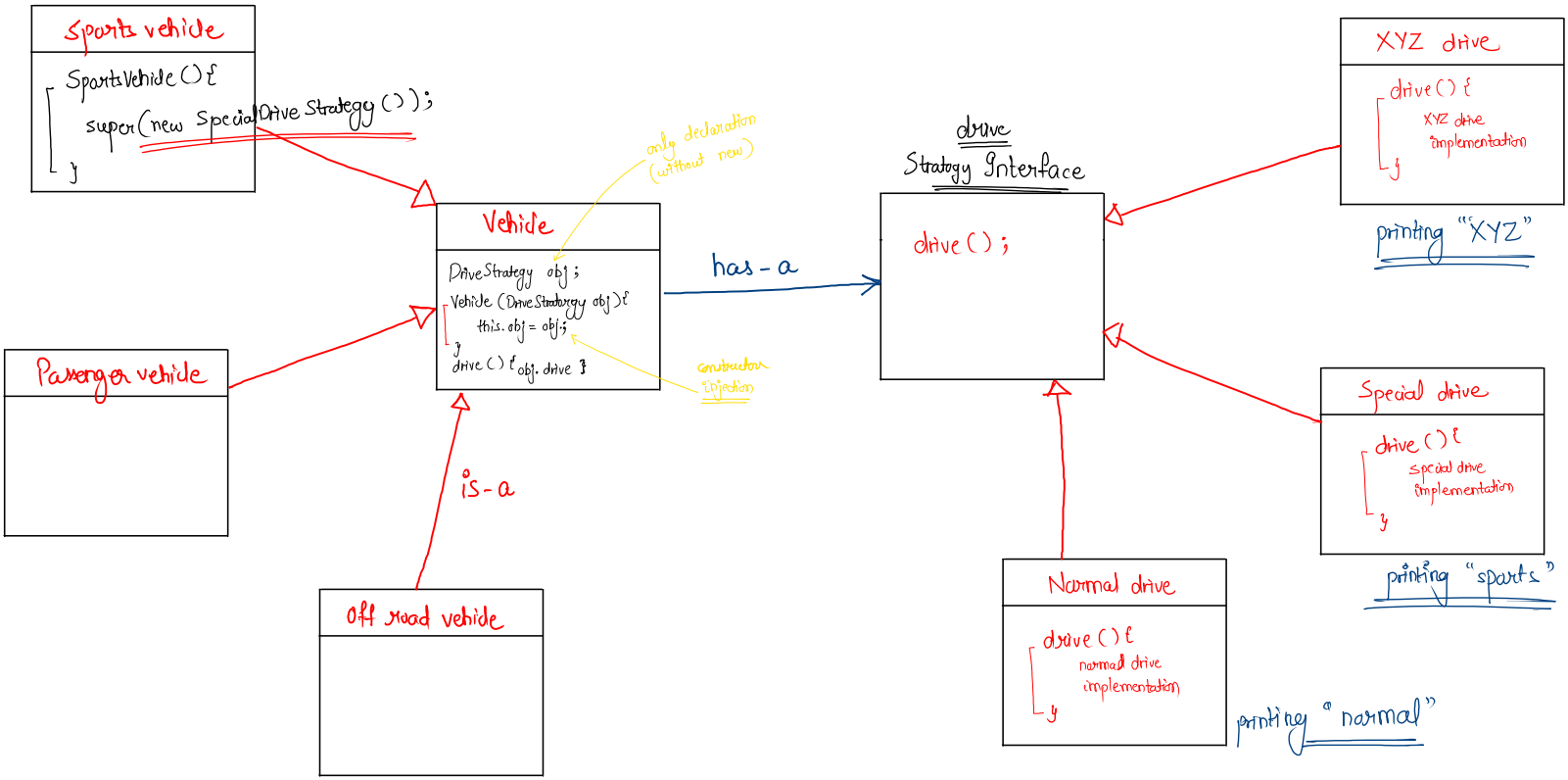
Ex:- Problem :-

Note :- → is-a (means inheritance)

→ has-a relationship (means a class has obj of other class)



Solution:- Now we have DriveStrategy Interface, which is implemention 3 type of drive and we can use any of it while implemention Passenger, Offroad or Sports Vehicle.



Note:- We just need to pass the object (let's say SportsVehicle type) to Vehical (parent class) using super() and it will intialize the obj with that type

Client code

```
public class Main {
    public static void main(String[] args){
        Vehicle obj = new SportsVehicle();
        obj.drive(); // print sports

        Vehicle obj = new NormalDrive();
        obj.drive(); // print normal
    }
}
```

⇒ Observer Design Pattern :- (Walmart interview question)

Que)

amazon.com

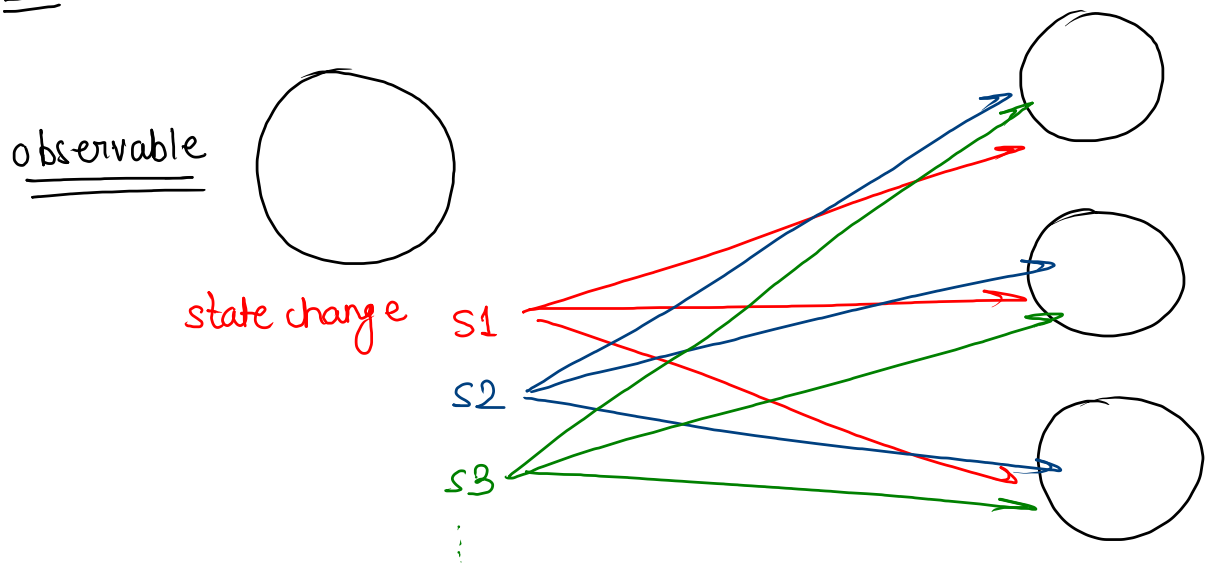
product is unavailable

notify me button

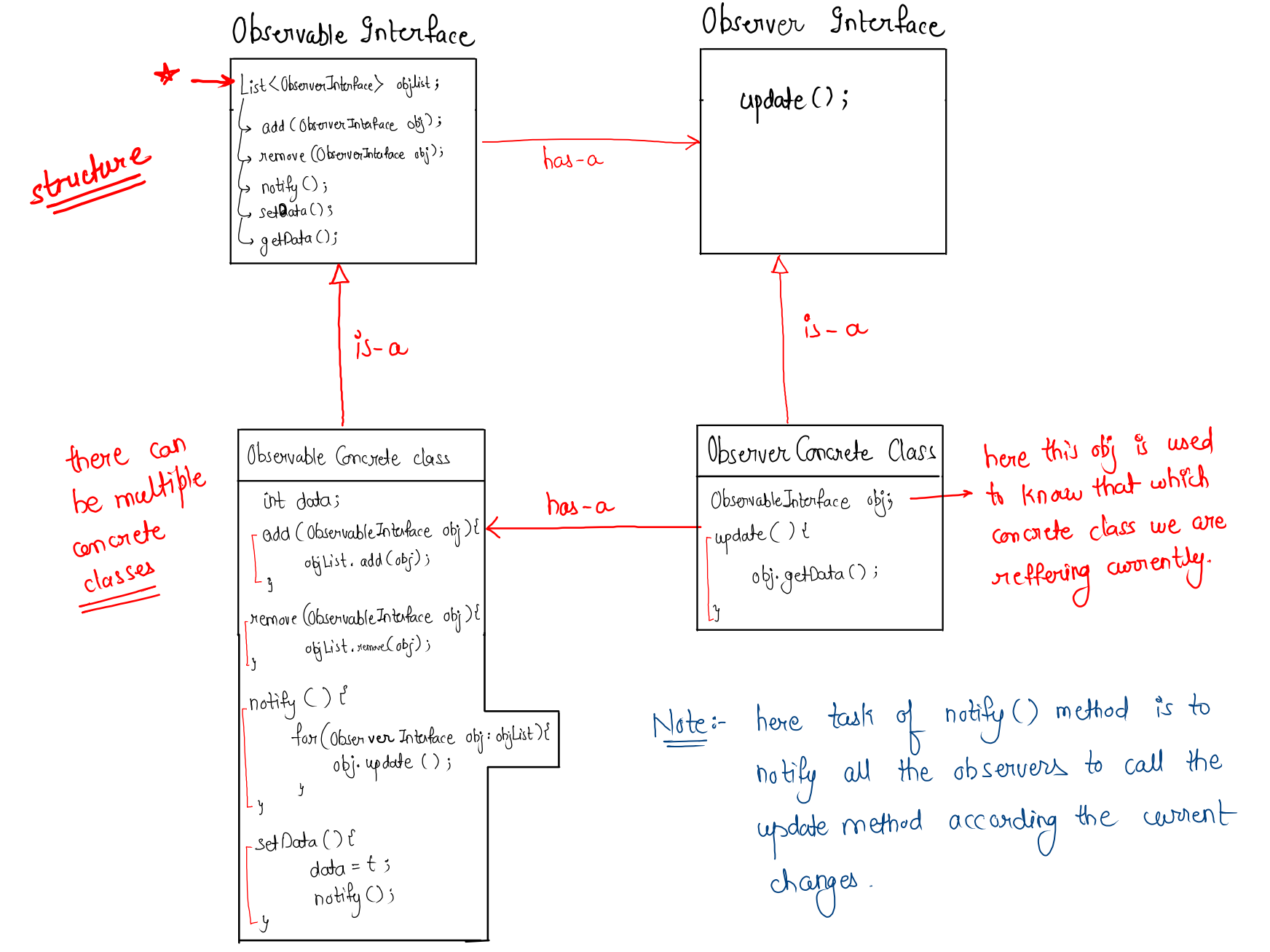
In amazon.com, we are looking for a product which is unavailable and there is a button of notify me!, so send notifications to customers when product is available.

Implement this button (LLD questions)

Note:- There are 2 states



whenever a state changes of observable then all observers will be notified

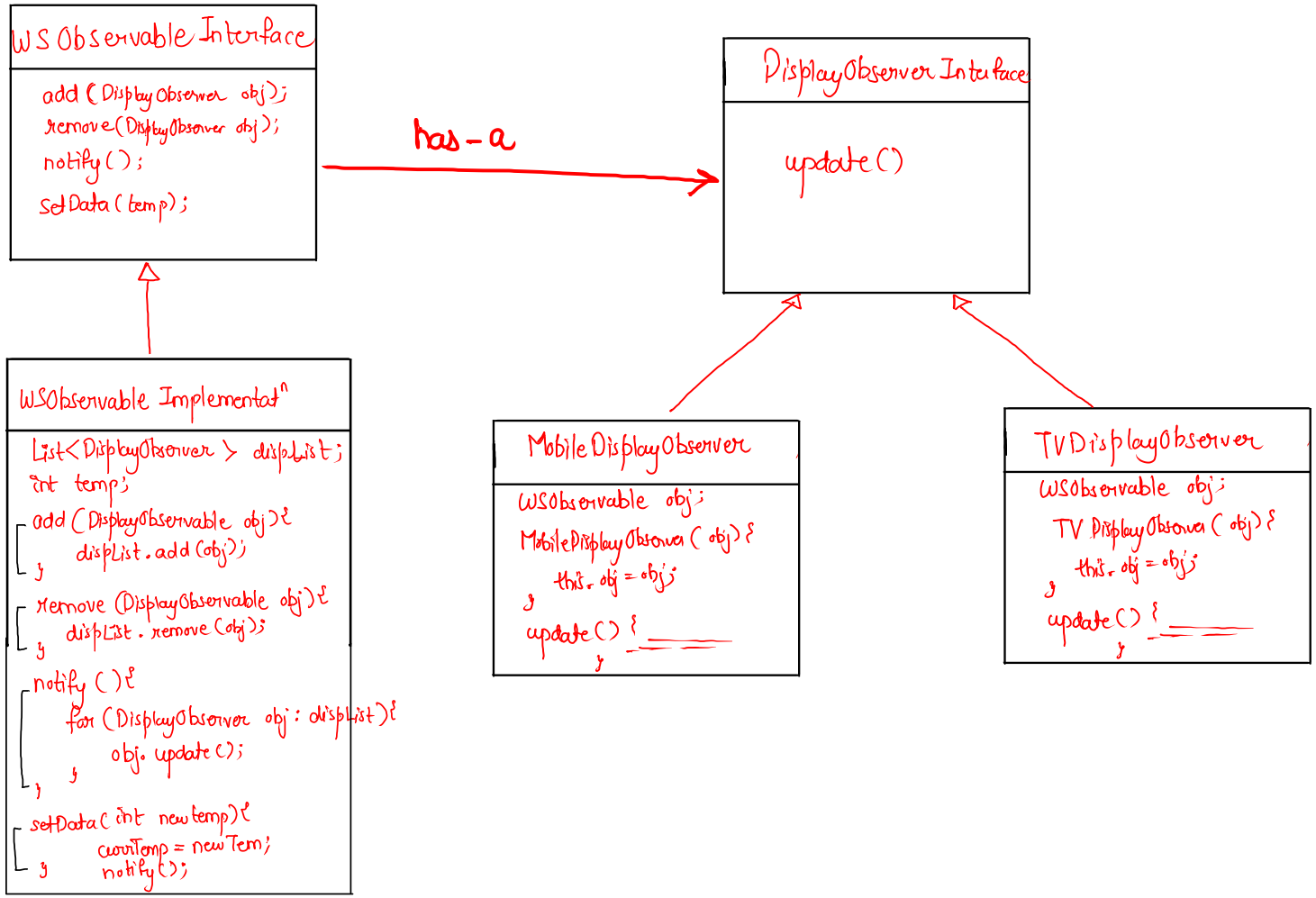

Observable Interface
List<ObserverInterface> objList;
add(ObserverInterface obj);
remove(ObserverInterface obj);
notify();
setData();
getData();
Observer Interface
update();
has-a
structure
is-a
is-a
there can be multiple concrete classes
Observable Concrete class
int data;
add(ObservableInterface obj){
objList.add(obj);
}
remove(ObservableInterface obj){
objList.remove(obj);
}
notify(){
for(ObserverInterface obj:objList){
obj.update();
}
}
setData(){
data = t;
notify();
}
has-a
Observer Concrete Class
ObservableInterface obj;
update(){
obj.getData();
}
here this obj is used to know that which concrete class we are reffering currently.
Note:- here task of notify() method is to notify all the observers to call the update method according the current changes.

Example

A Weather station is updating current temp. every hour → observed by → TV display observer, Mobile display observer

Solution

Note :- WSObservable = weather station observable



Solution of Walmart Interview Question :-

```java
public interface StocksObservable {

    public void add(NotificationAlertObserver observer);

    public void remove(NotificationAlertObserver observer);

    public void notifySubscribers();

    public void setStockCount(int newStockAdded);

    public int getStockCount();
}
```

has-a

```java
public interface NotificationAlertObserver {

    public void update();
}
```

```java
public class IphoneObservableImpl implements StocksObservable{

    public List<NotificationAlertObserver> observerList = new ArrayList<>();
    public int stockCount = 0;

    @Override
    public void add(NotificationAlertObserver observer) { observerList.add(observer); }

    @Override
    public void remove(NotificationAlertObserver observer) { observerList.remove(observer); }

    @Override
    public void notifySubscribers() {

        for(NotificationAlertObserver observer : observerList) {
            observer.update();
        }
    }

    public void setStockCount(int newStockAdded) {
        if(stockCount == 0) {
            notifySubscribers();
        }
        stockCount = stockCount + newStockAdded;
    }

    public int getStockCount() { return stockCount; }
}
```

```java
public class EmailAlertObserverImpl implements NotificationAlertObserver {

    String emailId;
    StocksObservable observable;

    public EmailAlertObserverImpl(String emailId, StocksObservable observable){

        this.observable = observable;
        this.emailId = emailId;
    }

    @Override
    public void update() {
        sendMail(emailId, msg: "product is in stock hurry up!");
    }

    private void sendMail(String emailId, String msg){

        System.out.println("mail sent to:" + emailId);
        //send the actual email to the end user
    }
}
```

```java
public class MobileAlertObserverImpl implements NotificationAlertObserver{

    String userName;
    StocksObservable observable;

    public MobileAlertObserverImpl(String emailId, StocksObservable observable){

        this.observable = observable;
        this.userName = emailId;
    }

    @Override
    public void update() { sendMsgOnMobile(userName, msg: "product is in stock hurry up!"); }

    private void sendMsgOnMobile(String userName, String msg){

        System.out.println("msg sent to:" + userName);
        //send the actual email to the end user
    }
}
```

Note:- here, we have created a stockobservable which we are implementing using IphoneObservableImpl, and now we want to notify the update to all the required customer, for which we have 2 type :- either we can sent it through mobile phone or we can send it through email.

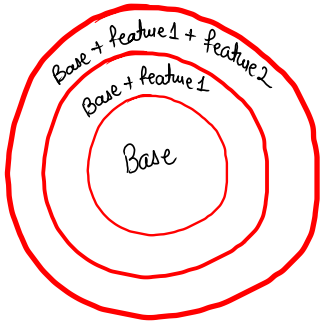
client code

```
public class Store {

    public static void main(String args[]) {

        StocksObservable iphoneStockObservable = new IphoneObservableImpl();

        NotificationAlertObserver observer1 = new EmailAlertObserverImpl( emailId: "xyz1@gmail.com", iphoneStockObservable);
        NotificationAlertObserver observer2 = new EmailAlertObserverImpl( emailId: "xyz2@gmail.com", iphoneStockObservable);
        NotificationAlertObserver observer3 = new MobileAlertObserverImpl( emailId: "xyz_username", iphoneStockObservable);

        iphoneStockObservable.add(observer1);
        iphoneStockObservable.add(observer2);
        iphoneStockObservable.add(observer3);

        iphoneStockObservable.setStockCount(10);

    }
}
```

this will notify all through email or mobile and update stock count by +10

⇒ Decoratar Design Pattern

(Ex:- Coffee machine design, pizza design, etc — Imp Question)

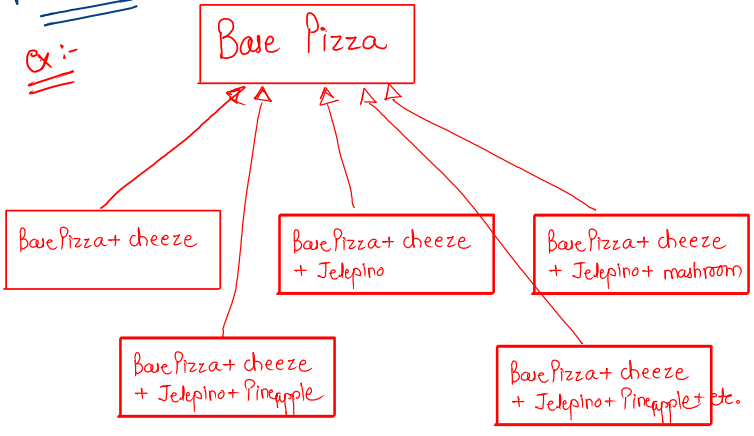


Why do we need decaratar pattern?

↳ To avoid class explosion

means:-

ex:-



This is where decaratar pattern comes into picture where base is same and we can keep adding features on top of it which will also work like a base for another feature to be added on top of it.

like, that how many different classes are we going to make with different combination, so it will be very difficult to manage.

→ famous example (Note:- a decorator is both is-a & has-a which is why it able to create many layers of objects)

has-a

```java
public abstract class BasePizza {

    public abstract int cost();
}
```

is-a

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

```java
public class Farmhouse extends BasePizza{

    @Override
    public int cost() {
        return 200;
    }
}
```

```java
public class VegDelight extends BasePizza{

    @Override
    public int cost() {
        return 120;
    }
}
```

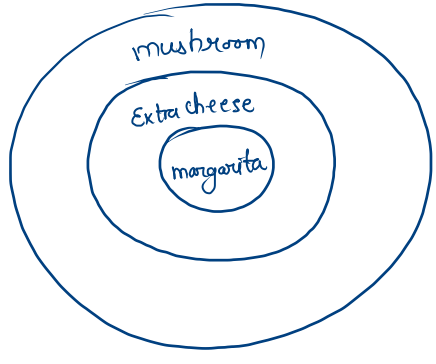
```java
public class Margherita extends BasePizza{

    @Override
    public int cost() {
        return 100;
    }
}
```

```java
public class ExtraCheese extends ToppingDecorator{

    BasePizza basePizza;

    public ExtraCheese(BasePizza pizza) {
        this.basePizza = pizza;
    }

    @Override
    public int cost() {
        return this.basePizza.cost() + 10;
    }
}
```

```java
public class Mushroom  extends ToppingDecorator{

    BasePizza basePizza;

    public Mushroom(BasePizza pizza) {
        this.basePizza = pizza;
    }

    @Override
    public int cost() {
        return this.basePizza.cost() + 15;
    }
}
```

Client code

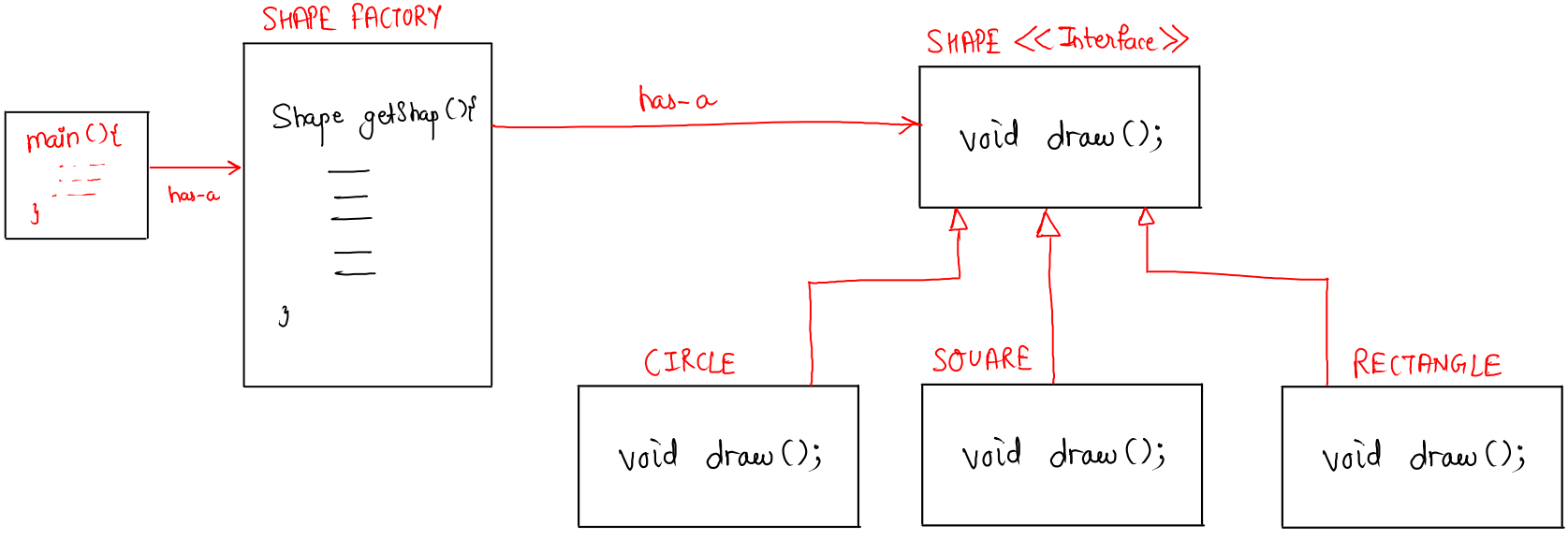
```java
Pizza pizza = new ExtraCheese(new Margarita());
pizza.cost()  // 100 + 10 = 110
```




```java
Pizza pizza = new Mushroom(new ExtraCheese(new Margarita()));  //3 decorators
pizza.cost()  // 100 + 10 + 15 = 125
```

⇒ Factory Pattern V. Imp

↳ Factory pattern provides an interface for creating objects in a superclass while allowing subclass to specify the type of object they create.



Example code

```java
public class MainClass {

    public static void main(String args[]) {

        ShapeFactory shapeFactoryObj = new ShapeFactory();
        Shape shapeObj = shapeFactoryObj.getShape( input: "CIRCLE");
        shapeObj.draw();
    }
}
```

```java
public class ShapeFactory {

    Shape getShape(String input) {

        switch (input) {

            case "CIRCLE":
                return new Circle();
            case "RECTANGLE":
                return new Rectangle();
            default:
                return null;
        }
    }
}
```

```java
public interface Shape {

    void draw();
}
```

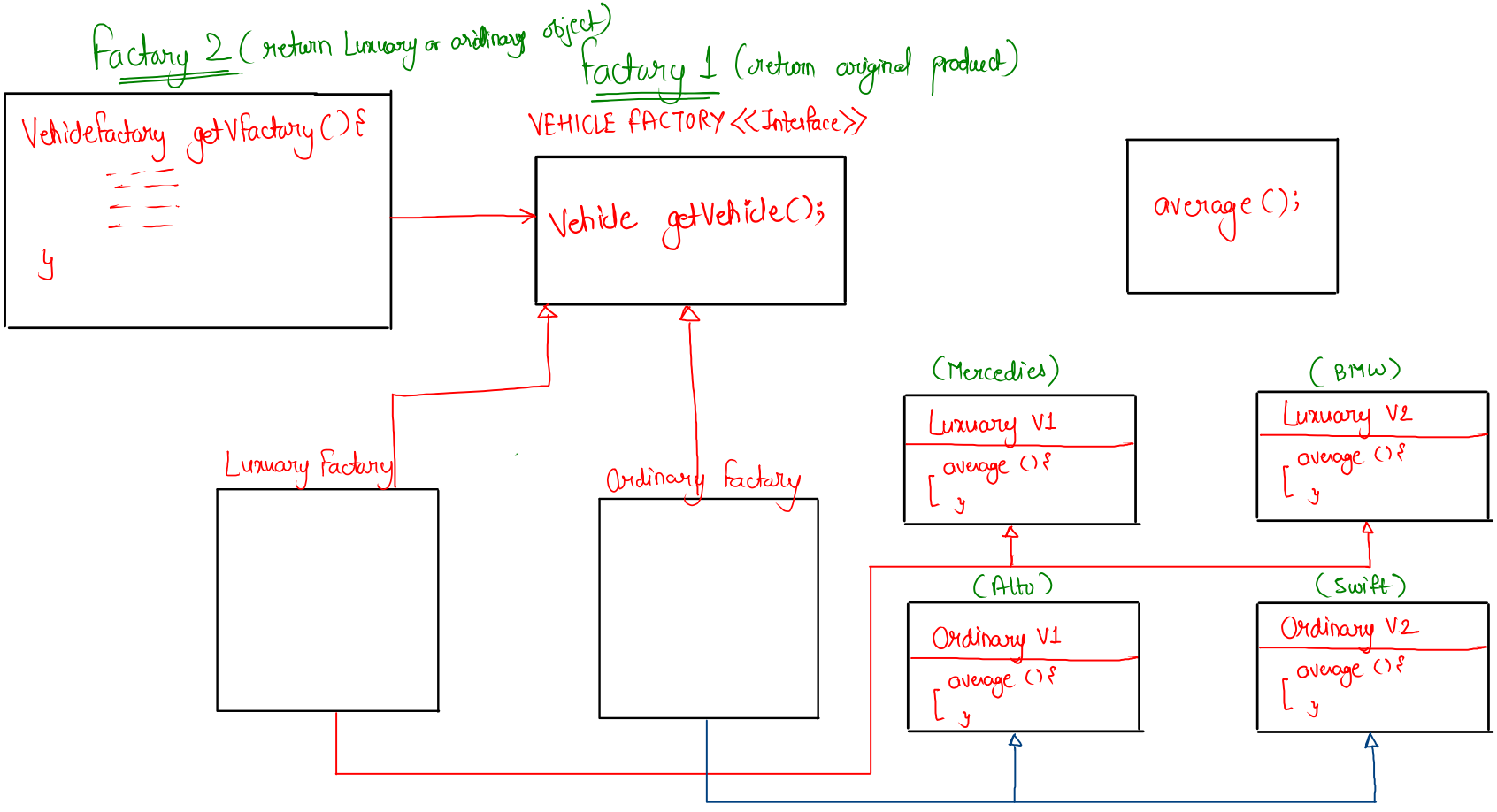
```java
public class Rectangle implements Shape{

    @Override
    public void draw() {
        System.out.println("rectangle");
    }
}
```

```java
public class Circle implements Shape{

    @Override
    public void draw() {

        System.out.println("cicle");
    }
}
```

Note:- we might need to create same object in many places in some cond?, in that scenario to avoid duplicacy we use factory design pattern

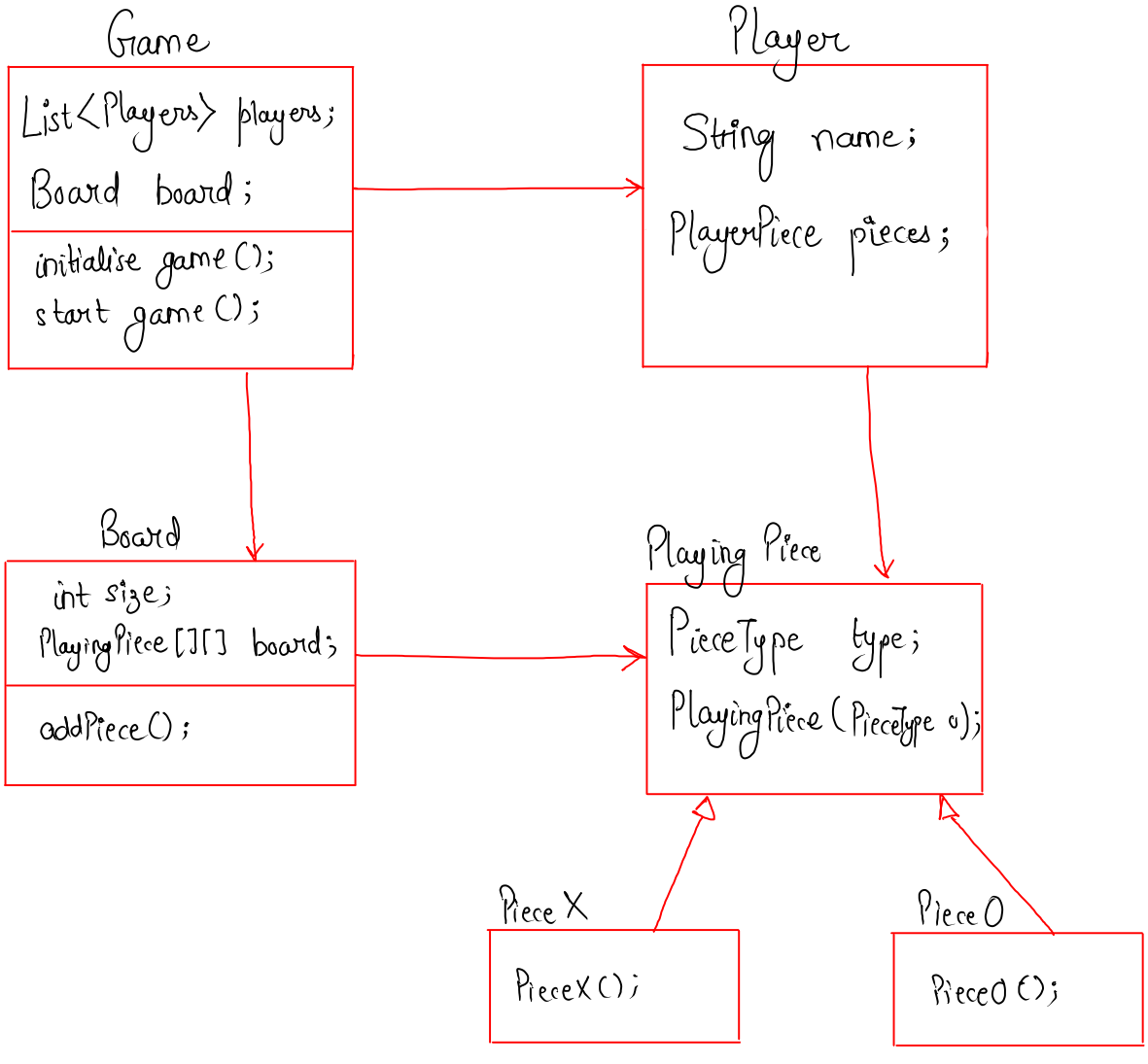
⇒ Abstract Factory Pattern :- (It's a Factory of factory)

↳ we can use this pattern, when we have many different patterns and we can group them saperately.



Quel Design Tic-Tac-Toe Game

UML Diagram:-




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

Complete code

Game

```java
public class Board {

    public int size;
    public PlayingPiece[][]board;

    public Board(int size) {
        this.size = size;
        board = new PlayingPiece[size][size];
    }


    public boolean addPiece(int row, int column, PlayingPiece playingPiece) {

        if(board[row][column] != null) {
            return false;
        }
        board[row][column] = playingPiece;
        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> rowColumn = new Pair<>(i, j);
                    freeCells.add(rowColumn);
                }
            }
        }

        return freeCells;
    }
    public void printBoard() {

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (board[i][j] != null) {
                   System.out.print(board[i][j].pieceType.name() + "   ");
                } else {
                    System.out.print("    ");

                }
                System.out.print(" | ");
            }
            System.out.println();

        }
    }
}
```

```java
public class Player {

    public String name;
    public PlayingPiece playingPiece;

    public Player(String name, PlayingPiece playingPiece) {
        this.name = name;
        this.playingPiece = playingPiece;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public PlayingPiece getPlayingPiece() {
        return playingPiece;
    }

    public void setPlayingPiece(PlayingPiece playingPiece) {
        this.playingPiece = playingPiece;
    }
}
```

```java
public class PlayingPiece {

    public PieceType pieceType;

    PlayingPiece(PieceType pieceType) {
        this.pieceType = pieceType;
    }
}
```

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

```java
public class PlayingPieceO extends PlayingPiece{

    public PlayingPieceO() {
        super(PieceType.O);
    }
}
```

```java
public class PlayingPieceX extends PlayingPiece{

    public PlayingPieceX() {
        super(PieceType.X);
    }
}
```

Game :-

```
public class TicTacToeGame {

    Deque<Player> players;
    Board gameBoard;


    public void initializeGame(){

        //creating 2 Players
       players = new LinkedList<>();
        PlayingPieceX crossPiece = new PlayingPieceX();
        Player player1 = new Player("Player1", crossPiece);

        PlayingPieceO noughtsPiece = new PlayingPieceO();
        Player player2 = new Player("Player2", noughtsPiece);

        players.add(player1);
        players.add(player2);

        //initializeBoard
        gameBoard = new Board(3);
    }

    public String startGame(){

        boolean noWinner = true;
        while(noWinner){

            //take out the player whose turn is and also put the player in the list back
            Player playerTurn = players.removeFirst();

            //get the free space from the board
            gameBoard.printBoard();
            List<Pair<Integer, Integer>> freeSpaces =  gameBoard.getFreeCells();
            if(freeSpaces.isEmpty()) {
                noWinner = false;
                continue;
            }

            //read the user input
            System.out.print("Player:" + playerTurn.name + " Enter row,column: ");
            Scanner inputScanner = new Scanner(System.in);
            String s = inputScanner.nextLine();
            String[] values = s.split(",");
            int inputRow = Integer.valueOf(values[0]);
            int inputColumn = Integer.valueOf(values[1]);


            //place the piece
            boolean pieceAddedSuccessfully = gameBoard.addPiece(inputRow,inputColumn, playerTurn.playingPiece);
            if(!pieceAddedSuccessfully) {
                //player can not insert the piece into this cell, player has to choose another cell
                System.out.println("Incorredt possition chosen, try again");
                players.addFirst(playerTurn);
                continue;
            }
            players.addLast(playerTurn);

            boolean winner = isThereWinner(inputRow, inputColumn, playerTurn.playingPiece.pieceType);
            if(winner) {
                return playerTurn.name;
            }
        }

        return "tie";
    }

    public boolean isThereWinner(int row, int column, PieceType pieceType) {

        boolean rowMatch = true;
        boolean columnMatch = true;
        boolean diagonalMatch = true;
        boolean antiDiagonalMatch = true;

        //need to check in row
        for(int i=0;i<gameBoard.size;i++) {

            if(gameBoard.board[row][i] == null || gameBoard.board[row][i].pieceType != pieceType) {
                rowMatch = false;
            }
        }
        //need to check in column
        for(int i=0;i<gameBoard.size;i++) {

            if(gameBoard.board[i][column] == null || gameBoard.board[i][column].pieceType != pieceType) {
                columnMatch = false;
            }
        }

        //need to check diagonals
        for(int i=0, j=0; i<gameBoard.size;i++,j++) {
            if (gameBoard.board[i][j] == null || gameBoard.board[i][j].pieceType != pieceType) {
                diagonalMatch = false;
            }
        }

        //need to check anti-diagonals
        for(int i=0, j=gameBoard.size-1; i<gameBoard.size;i++,j--) {
            if (gameBoard.board[i][j] == null || gameBoard.board[i][j].pieceType != pieceType) {
                antiDiagonalMatch = false;
            }
        }

        return rowMatch || columnMatch || diagonalMatch || antiDiagonalMatch;
    }
}
```

Client code

main

```
public class Main {

    public static void main(String args[]) {
        TicTacToeGame game = new TicTacToeGame();
        game.initializeGame();
        System.out.println("game winner is: " + game.startGame());
    }
}
```

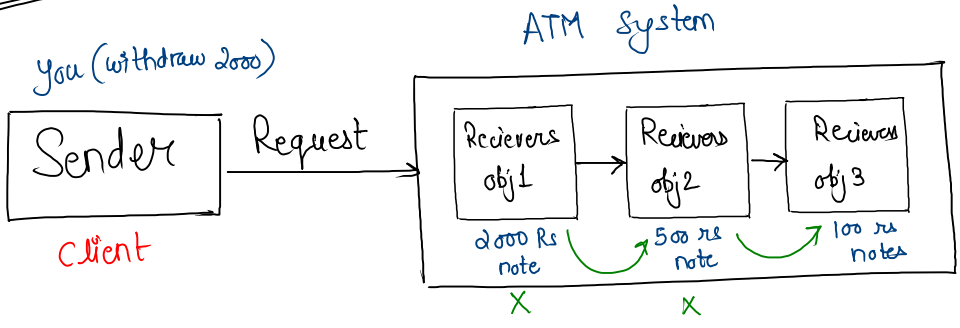
⇒ Chain of responsibility Design Pattern

Application usage :-

- ATM / Vending machine
- Design logger (Amazon)

structure

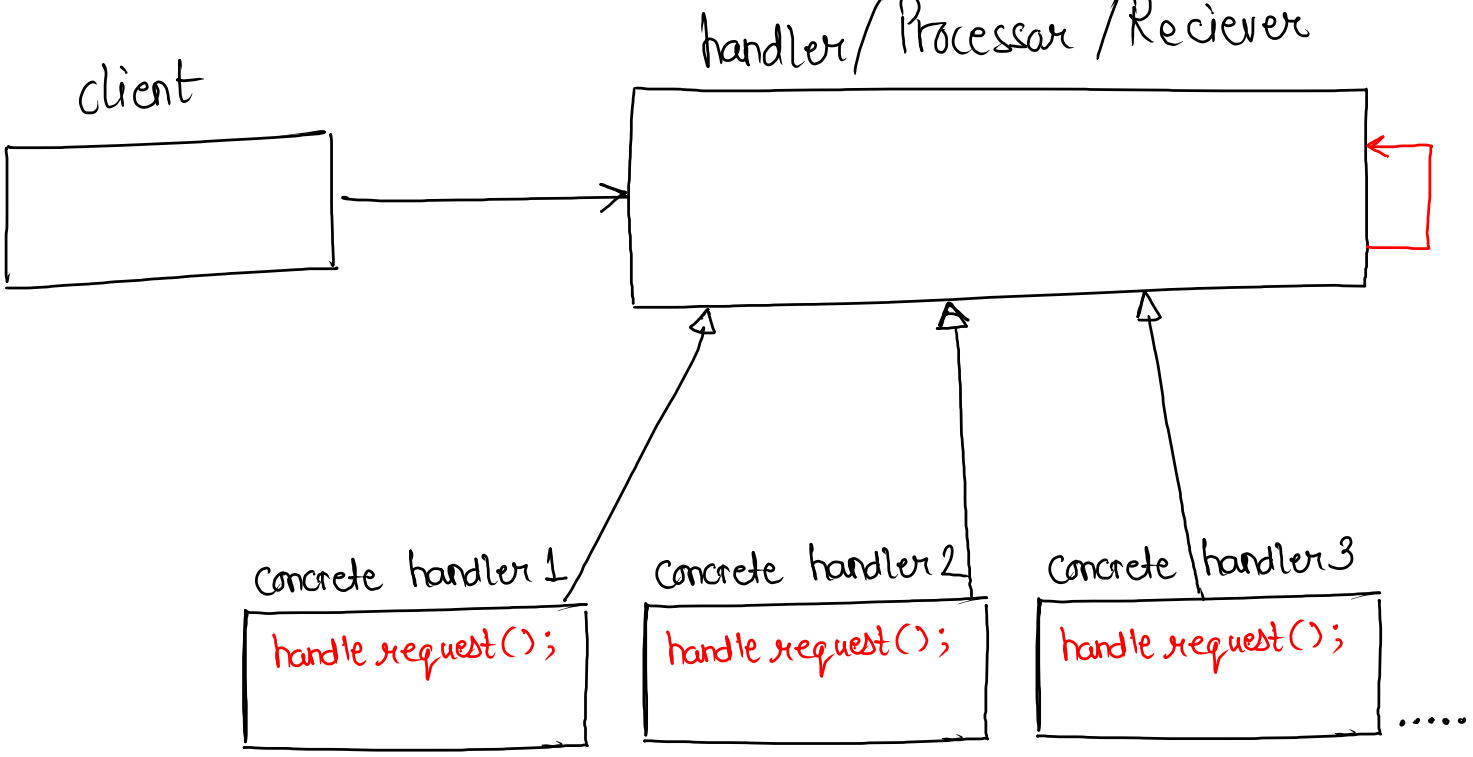
Ex:- You (withdraw 2000)



Note:- This type of design is used when we client sends a request and it doesn't matter that who is completing that request

Ex:-
in example, we want to withdraw 2000Rs, so we sended a request and if obj1 doesn't have enough amount then it sended the request for remaining amount to next object.
and if total amount is not enough then return insufficient amount.

Structure



Que) Design logger

```
Logger obj = new Logger();
obj.log(Info, "msg");
obj.log(Debug, "msg");
obj.log(Error, "msg");
```

Code

if info then print, else check next obj Debug and then error and lastly null

here this chaining is imp

```java
public class Main {

    public static void main(String args[]) {

        LogProcessor logObject = new InfoLogProcessor(new DebugLogProcessor(new ErrorLogProcessor( nexLogProcessor: null)));

        logObject.log(LogProcessor.ERROR,  message: "exception happens");
        logObject.log(LogProcessor.DEBUG,  message: "need to debug this ");
        logObject.log(LogProcessor.INFO,  message: "just for info ");

    }
}
```

here constructor is already storing next logger processor

```java
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);
        }
    }
}
```

```java
public class InfoLogProcessor extends LogProcessor{

    InfoLogProcessor(LogProcessor nexLogProcessor){
        super(nexLogProcessor);
    }

    public void log(int logLevel,String message){

        if(logLevel == INFO) {
            System.out.println("INFO: " + message);
        } else{

            super.log(logLevel, message);
        }

    }
}
```

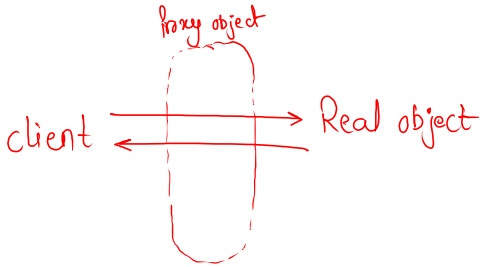
```java
public class ErrorLogProcessor extends LogProcessor{

    ErrorLogProcessor(LogProcessor nexLogProcessor) { super(nexLogProcessor); }

    public void log(int logLevel,String message){

        if(logLevel == ERROR) {
            System.out.println("ERROR: " + message);
        } else{

            super.log(logLevel, message);
        }

    }
}
```

```java
public class DebugLogProcessor extends LogProcessor{

    DebugLogProcessor(LogProcessor nexLogProcessor) { super(nexLogProcessor); }

    public void log(int logLevel,String message){

        if(logLevel == DEBUG) {
            System.out.println("DEBUG: " + message);
        } else{

            super.log(logLevel, message);
        }

    }
}
```

⇒ Proxy Design Pattern (very commonly used)

structure

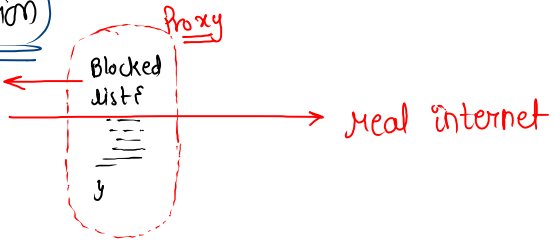
Ex1



If a client want to access the real object then proxy object is always inbetween and access the request before approving it.

(Internet Restriction)

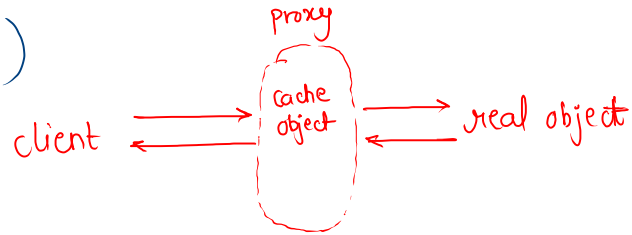
Ex2



here, when a user wants to access the internet, it passes through proxy which has a blocklist that which servers are blocked.

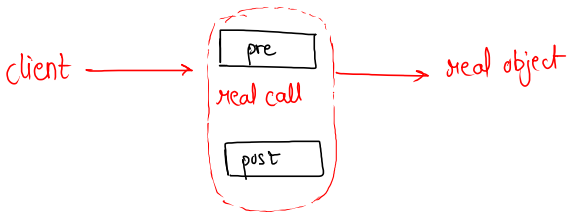
(Caching)

Ex3



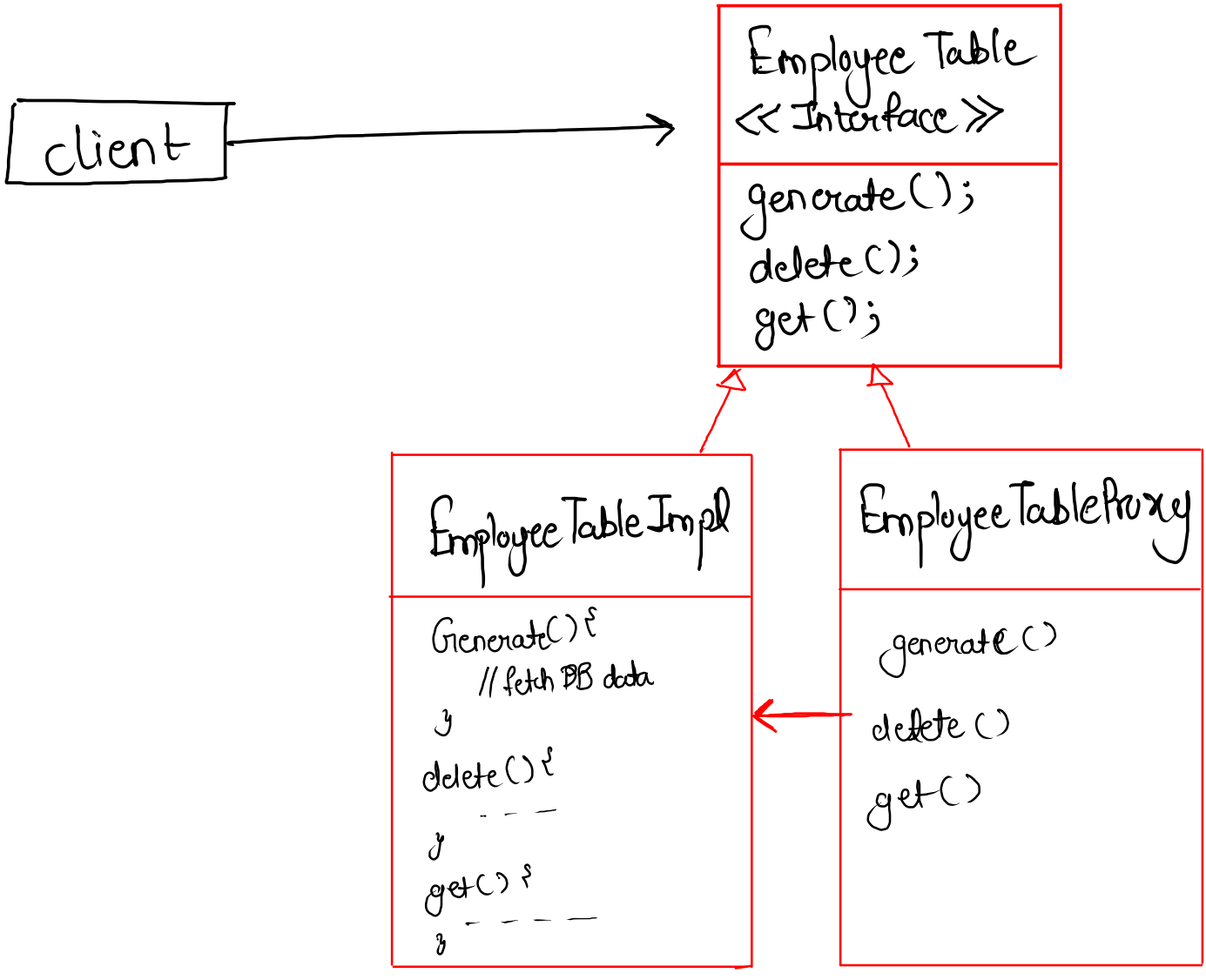
Caching is another example of proxy design pattern, because here as well we first have a proxy in b/w client & real object to check does cache already have the result.

Ex4 (Preprocessing & postprocessing)



If we want to perform any task before or after the calling is being done.

Structure


client
Employee Table
<< Interface >>
generate();
delete();
get();
EmployeeTableImpl
Generate(){
// fetch DB data
}
delete(){
- - -
}
get(){
- - - -
}
EmployeeTableProxy
generate()
delete()
get()

Code

```java
public class ProxyDesignPattern {

    public static void main(String args[]) {

        try {
            EmployeeDao empTableObj = new EmployeeDaoProxy();
            empTableObj.create( client: "USER", new EmployeeDo());
            System.out.println("Operation successful");

        }catch (Exception e) {
            System.out.println(e.getMessage());
        }

    }
}
```

```java
public interface EmployeeDao {

    public void create(String client, EmployeeDo obj) throws Exception;
    public void delete(String client, int employeeId) throws Exception;
    public EmployeeDo get(String client, int employeeId) throws Exception;
}
```

```java
public class EmployeeDaoImpl implements EmployeeDao{

    @Override
    public void create(String client, EmployeeDo obj) throws Exception {

        //creates a new Row
        System.out.println("created new row in the Employee table");
    }

    @Override
    public void delete(String client, int employeeId) throws Exception{

        //delete a Row
        System.out.println("deleted row with employeedID:" + employeeId);
    }

    @Override
    public EmployeeDo get(String client, int employeeId) throws Exception {

        //fetch row
        System.out.println("fetching data from the DB");
        return new EmployeeDo();
    }
}
```

has-a

```java
public class EmployeeDaoProxy implements EmployeeDao{

    EmployeeDao employeeDaoObj;

    EmployeeDaoProxy() {
        employeeDaoObj = new EmployeeDaoImpl();
    }

    @Override
    public void create(String client, EmployeeDo obj) throws Exception {

        if(client.equals("ADMIN")) {

            employeeDaoObj.create(client, obj);
            return;
        }

        throw new Exception("Access Denied");
    }

    @Override
    public void delete(String client, int employeeId) throws Exception{

        if(client.equals("ADMIN")) {

            employeeDaoObj.delete(client, employeeId);
            return;
        }

        throw new Exception("Access Denied");
    }

    @Override
    public EmployeeDo get(String client, int employeeId) throws Exception{

        if(client.equals("ADMIN") || client.equals("USER")) {

            return employeeDaoObj.get(client, employeeId);
        }
```

Imp

Note :- We can have as many Proxy as we want and proxyA will treat proxyB as a real object