Pre-requisite

- OOP's Concepts
- SOLID Principles

⇒ Design Patterns

design pattern are solution to common problems encounted by software developers when designing and building applications.

1.) Creational design pattern are concerned with the creation of objects. They provide ways to create object in a manner that is suitable for the situation in hand, without specifying the exact details of how the object is created.
(Factory, abstract factory, singleton, prototype, builder)

2) Structural design Pattern are concerned with composition of classes & objects. They provide flexible & efficient ways to combine classes & objects
(adapter, bridge, composite, decorator, facade, Flyweight, Proxy)

3) Behavioral design Pattern are concerned b/w the communications b/w objects. They provide ways to communicate b/w object and flow of information.
(Chain of responsibility, Command, Interpreter, mediator, observer, state, strategy, template, visitor)

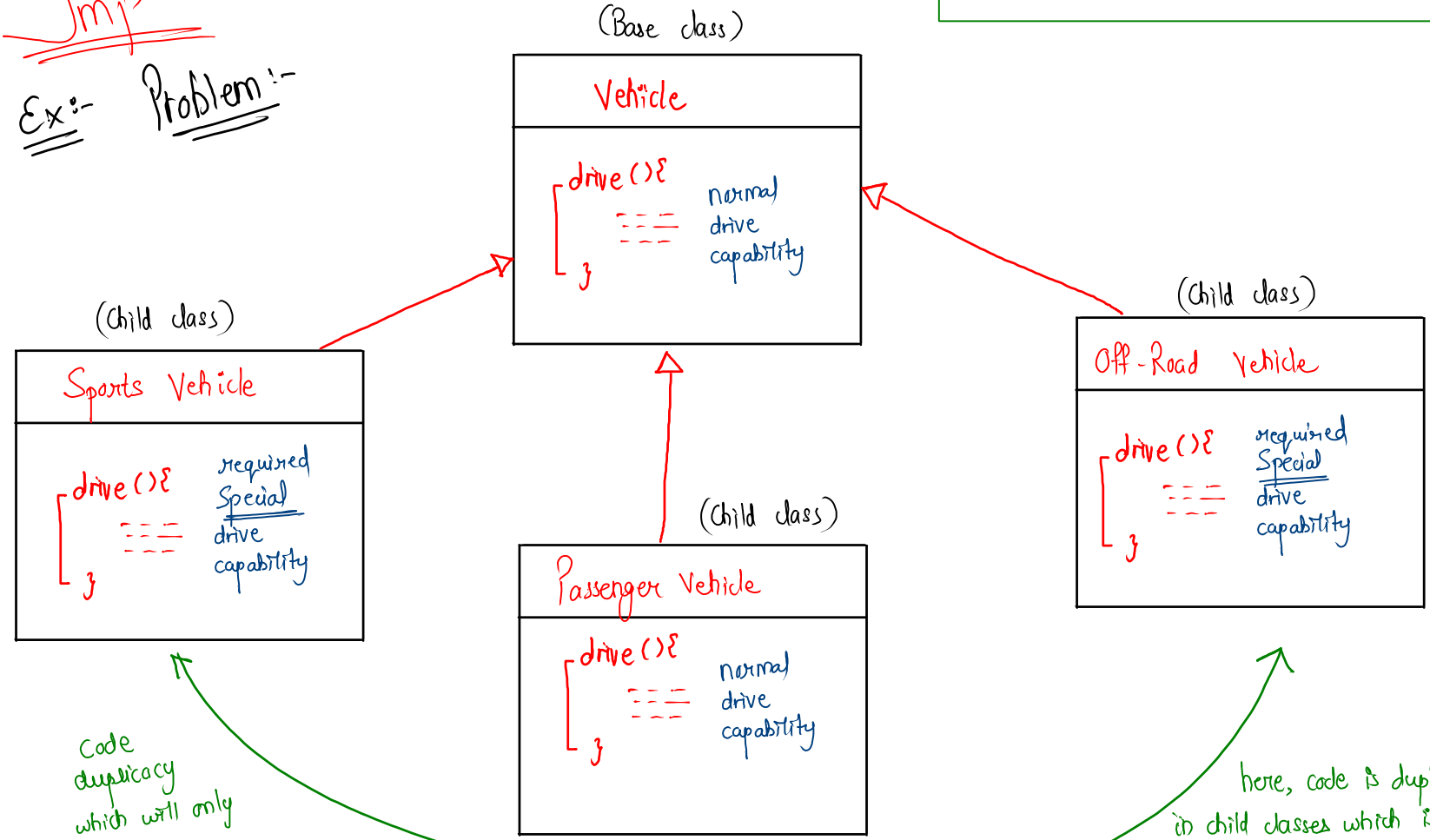
⇒ Strategy Design Pattern

Imp

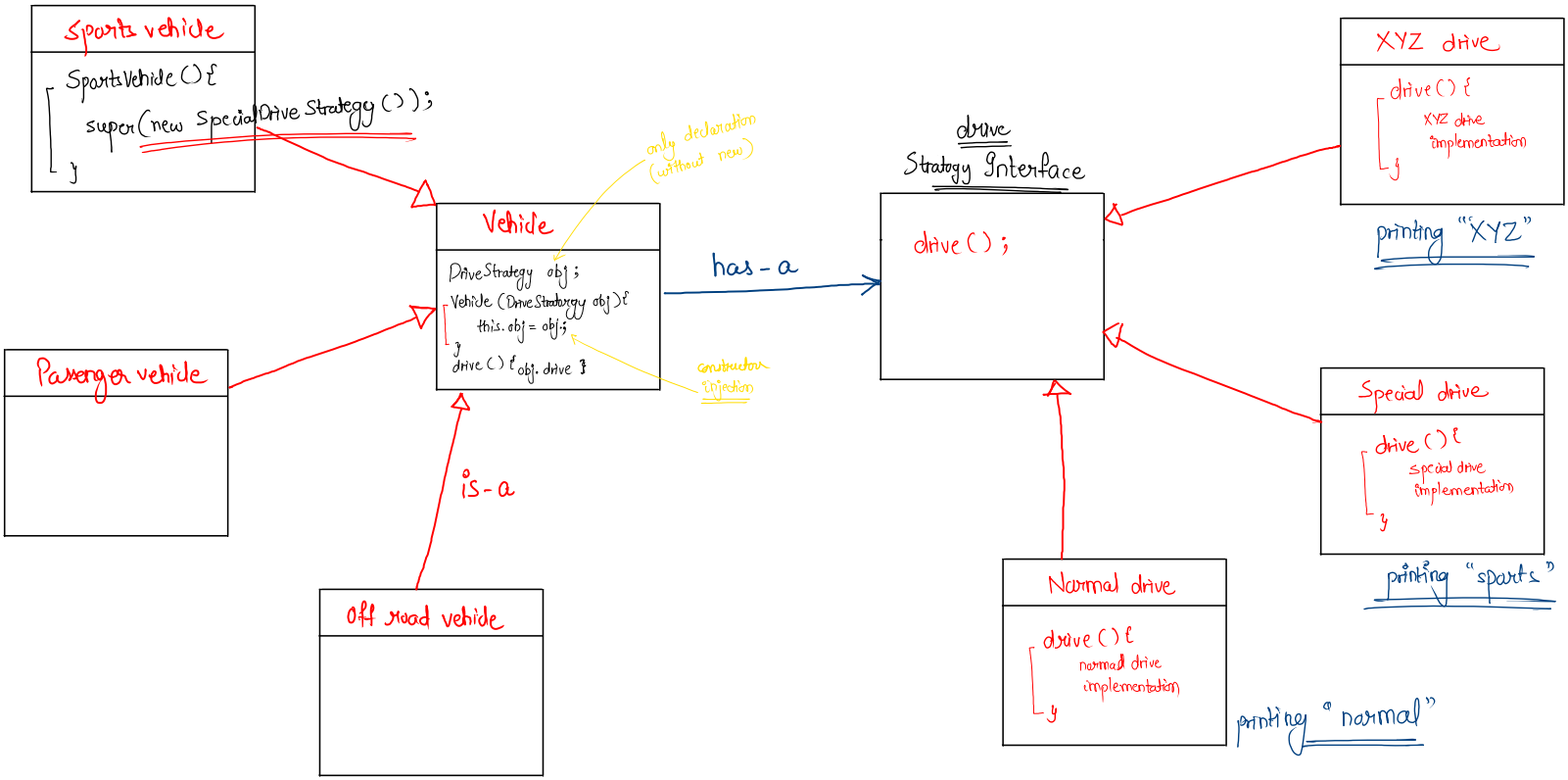
Ex:- Problem :-

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

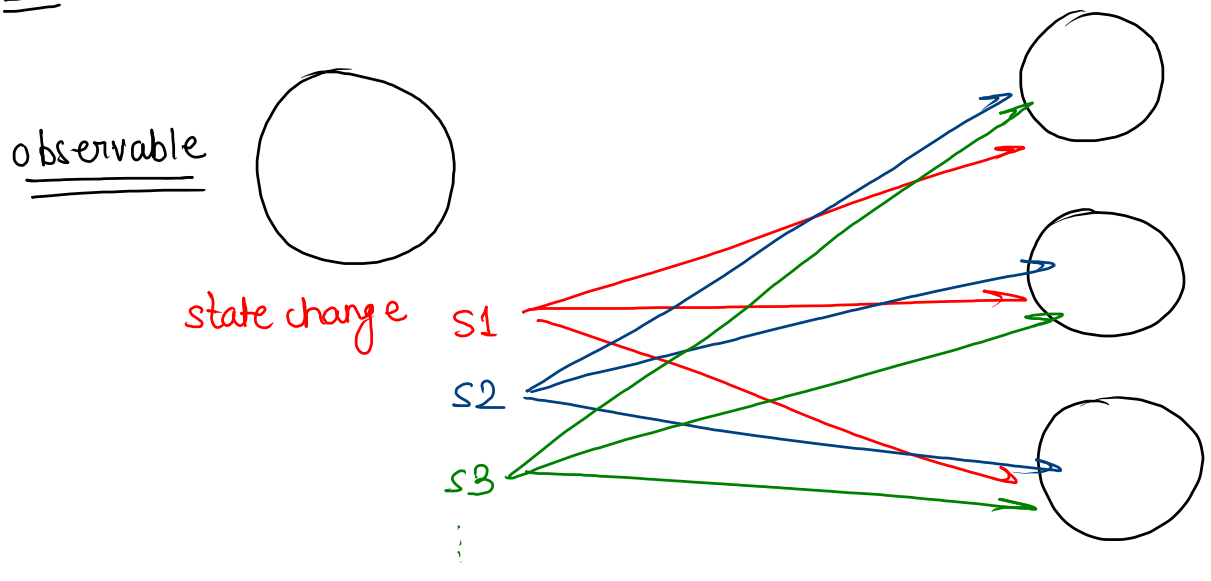
product is unavailable

notify me button

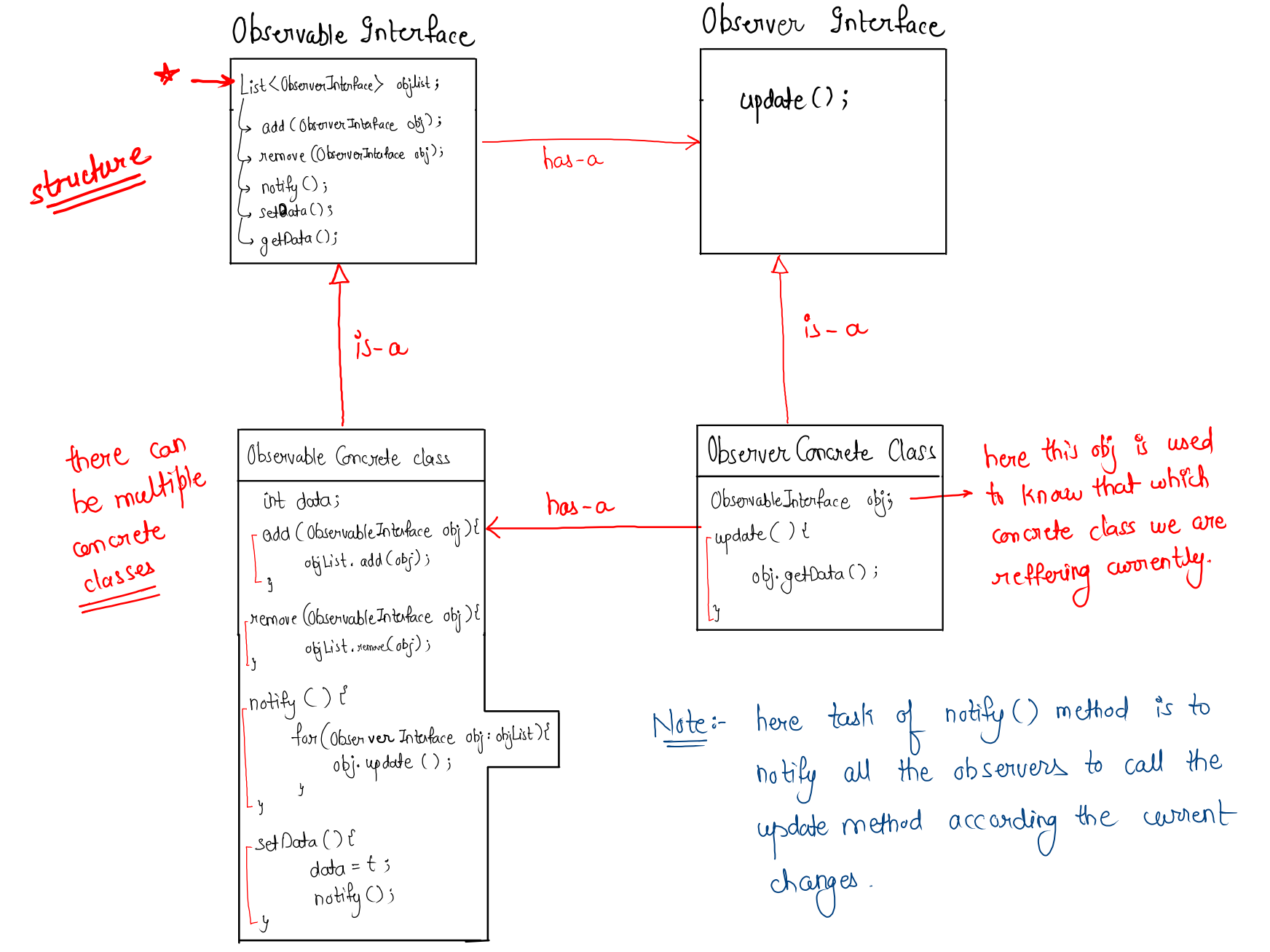
In amazon.com, we are looking for a product which is unavailable and there is a button of notify me!, so send notifications to customers when product is available.

Implement this button (LLD questions)

Note:- There are 2 states



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

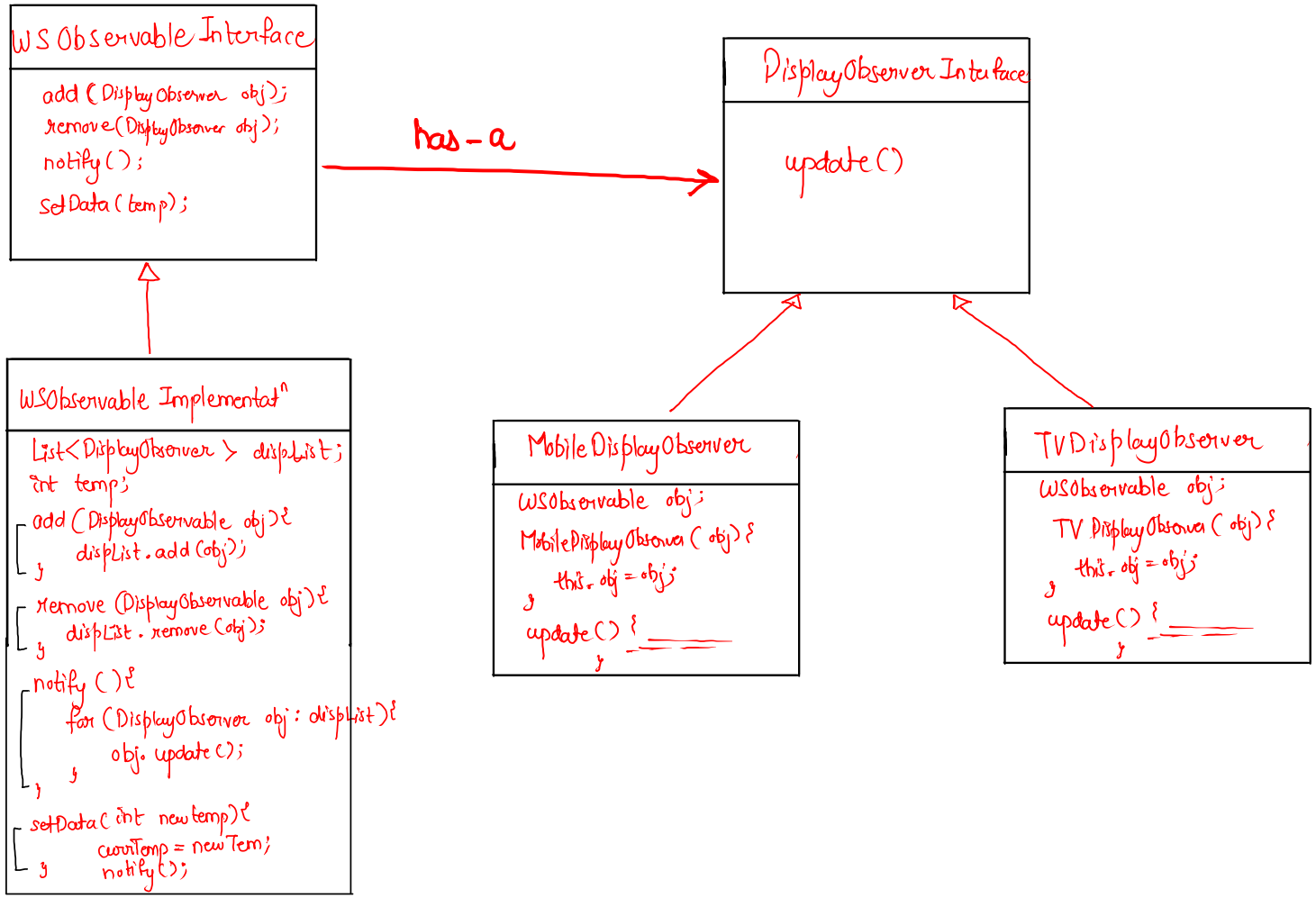

Observable Interface
List<ObserverInterface> objList;
add(ObserverInterface obj);
remove(ObserverInterface obj);
notify();
setData();
getData();
Observer Interface
update();
has-a
structure
is-a
is-a
there can be multiple concrete classes
Observable Concrete class
int data;
add(ObservableInterface obj){
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.

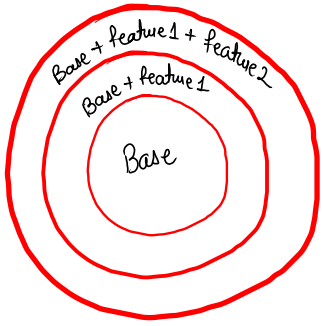
client code

```
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)

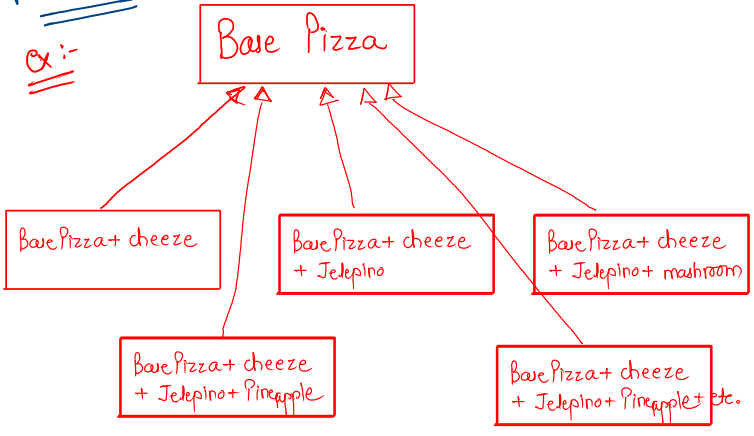


Why do we need decaratar pattern?

↳ 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;
    }
}
```

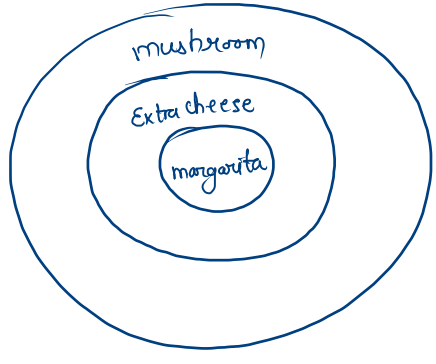
```java
public class Margherita extends BasePizza{

    @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

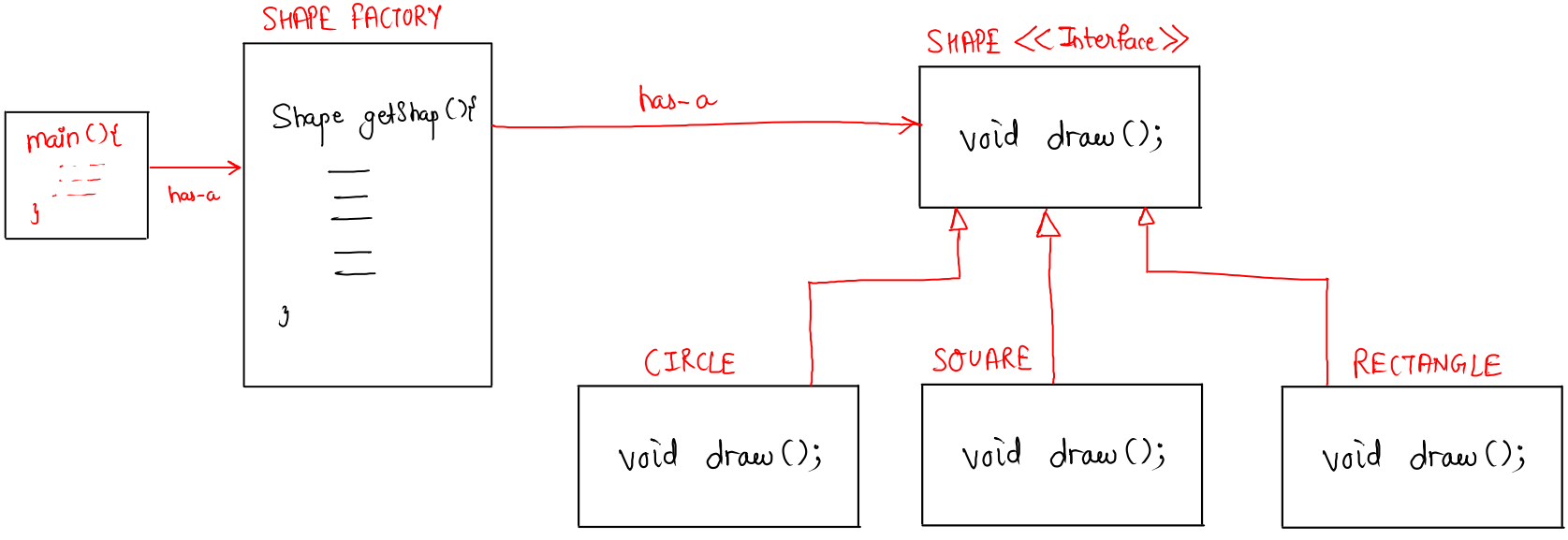
```java
Pizza pizza = new ExtraCheese(new Margarita());
pizza.cost()  // 100 + 10 = 110
```




```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();
}
```

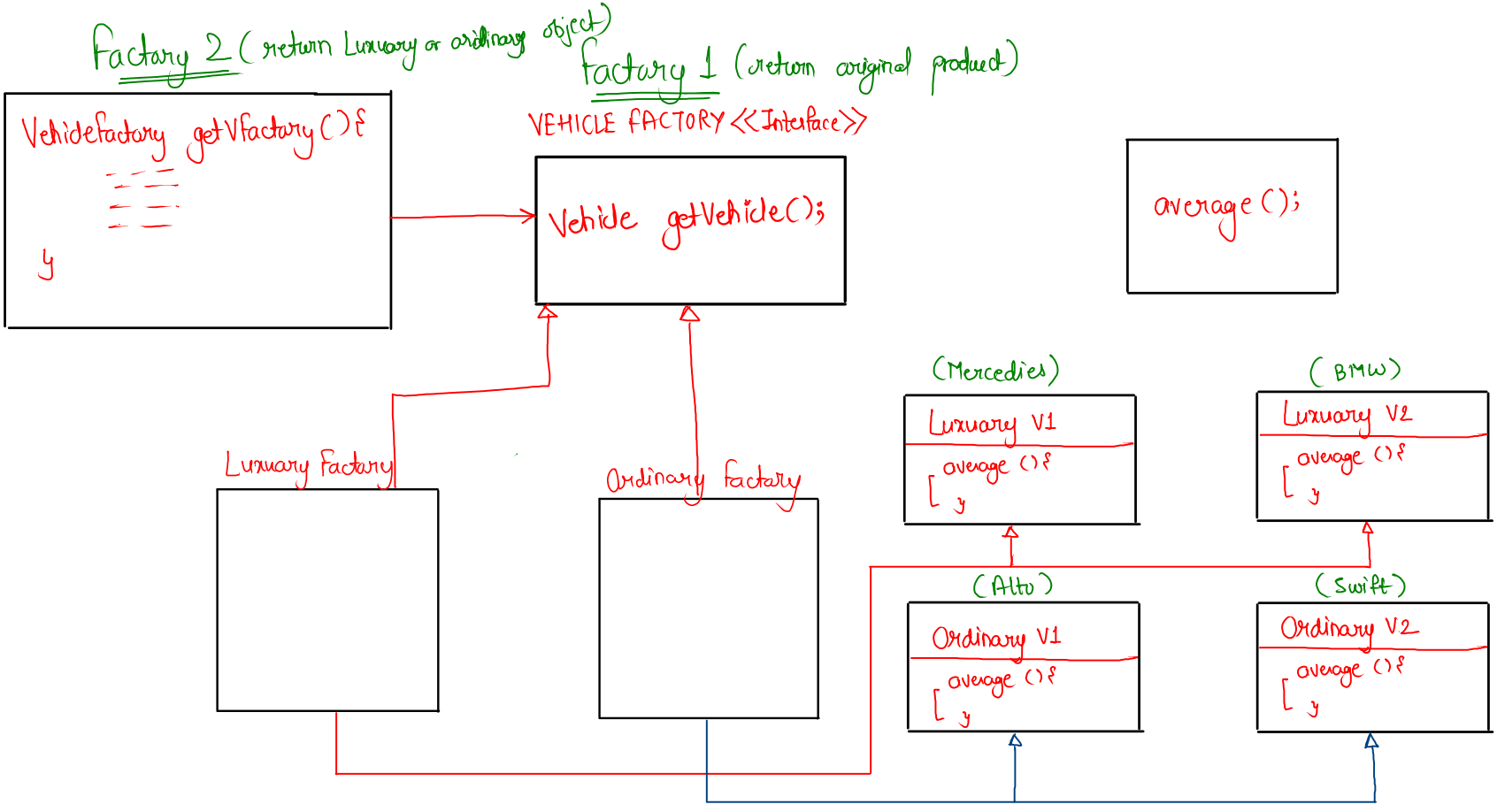
```java
public class Rectangle implements Shape{

    @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

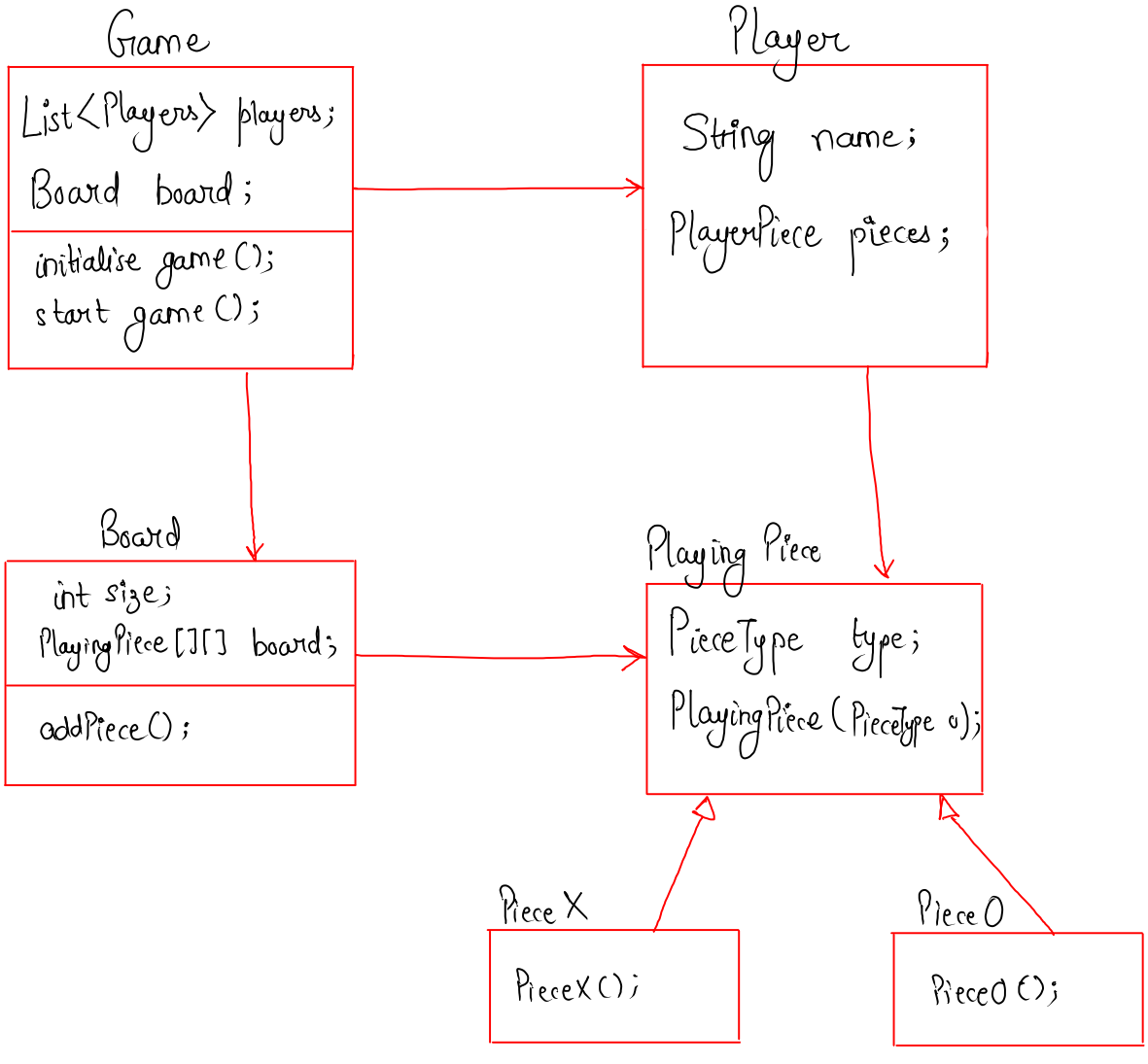
⇒ Abstract Factory Pattern :- (It's a Factory of factory)

↳ 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());
    }
}
```

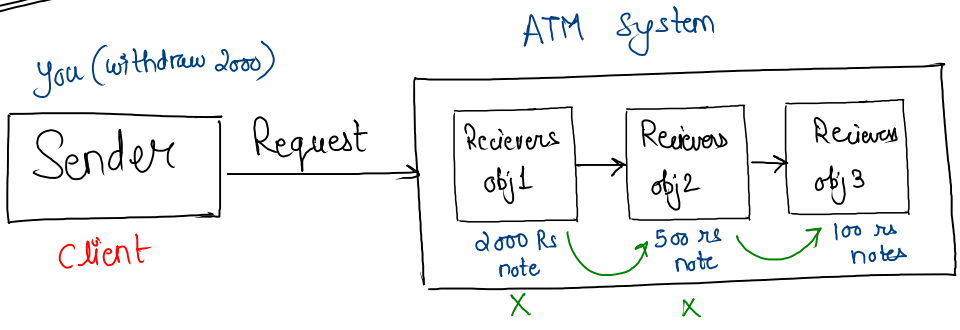
⇒ Chain of responsibility Design Pattern

Application usage :-

- ATM / Vending machine
- Design logger (Amazon)

structure

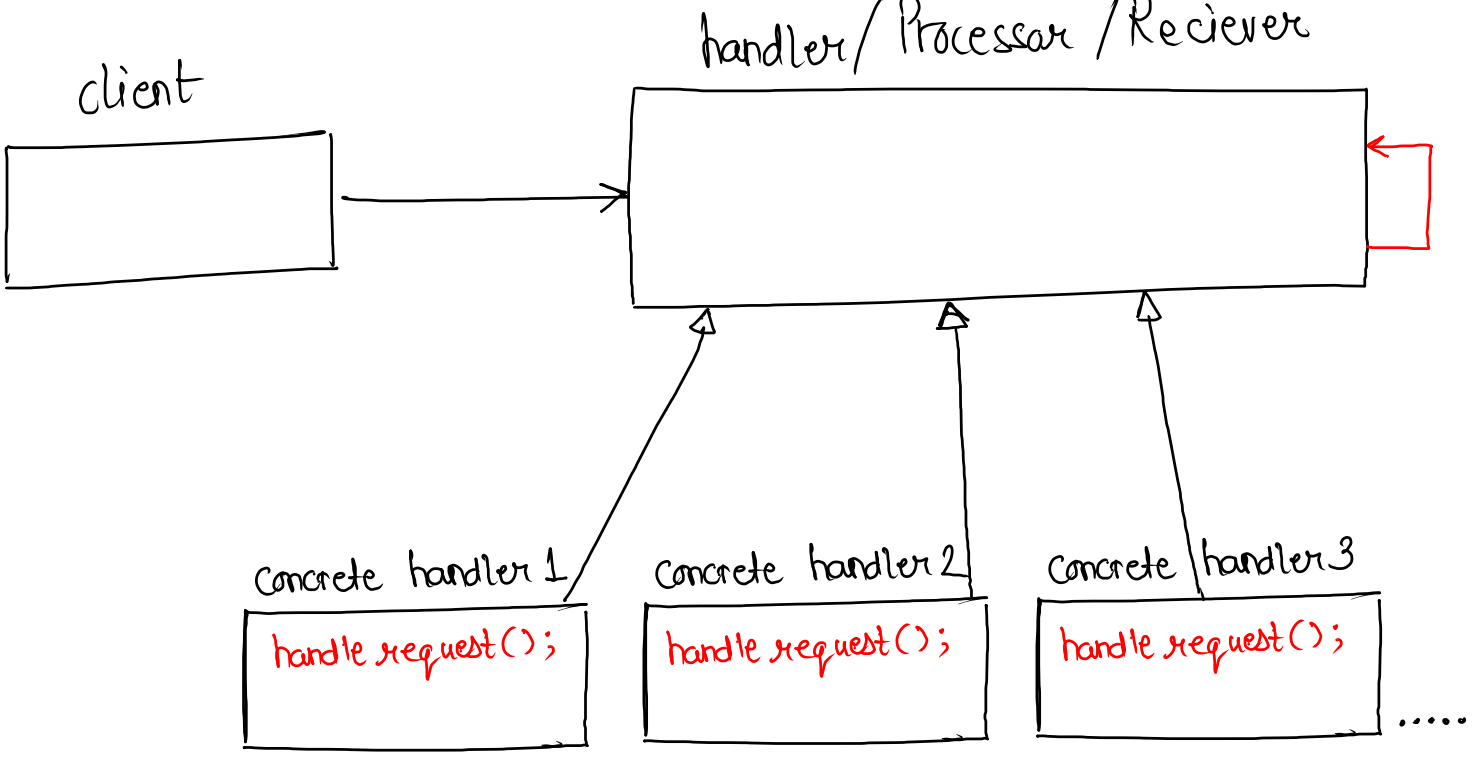
Ex:- You (withdraw 2000)



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);
        }

    }
}
```

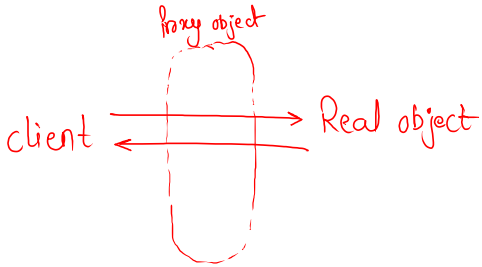
```java
public class ErrorLogProcessor extends LogProcessor{

    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

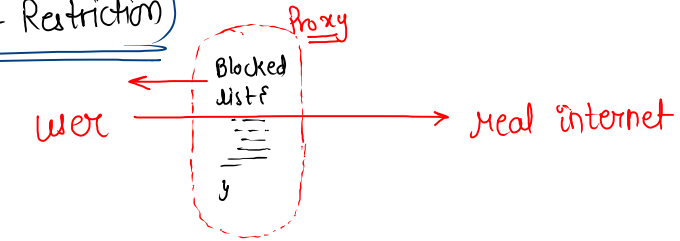
Ex1



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

(Internet Restriction)

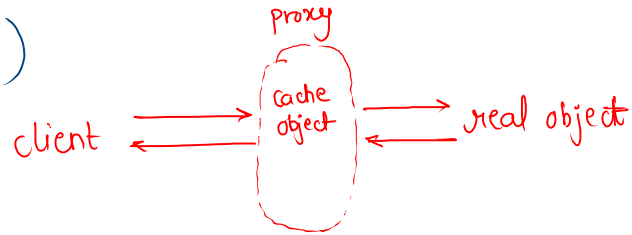
Ex2



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

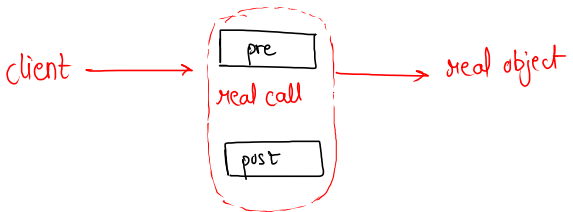
(Caching)

Ex3



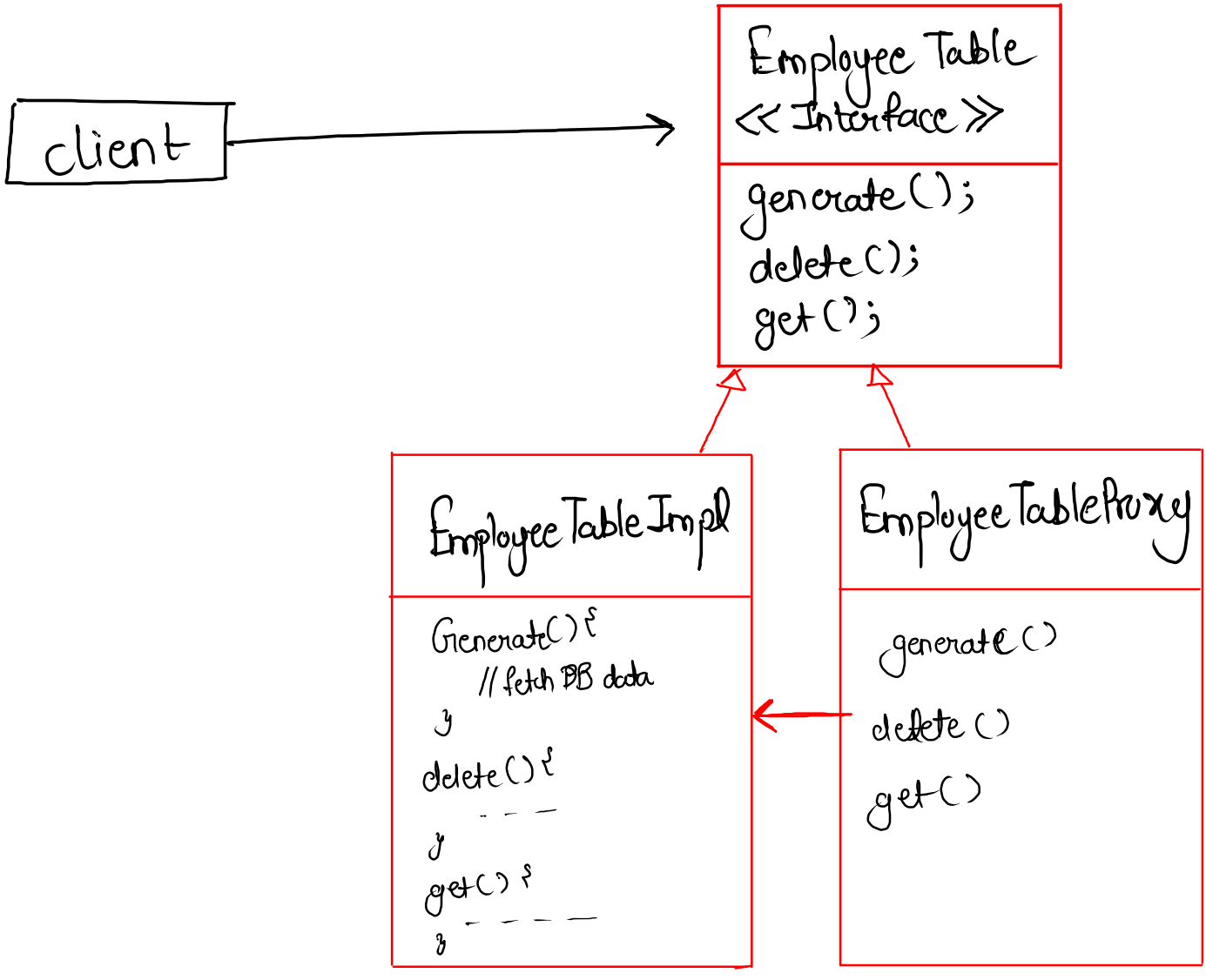
Caching is another example of proxy design pattern, because here as well we first have a proxy in b/w client & real object to check does cache already have the result.

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;
}
```

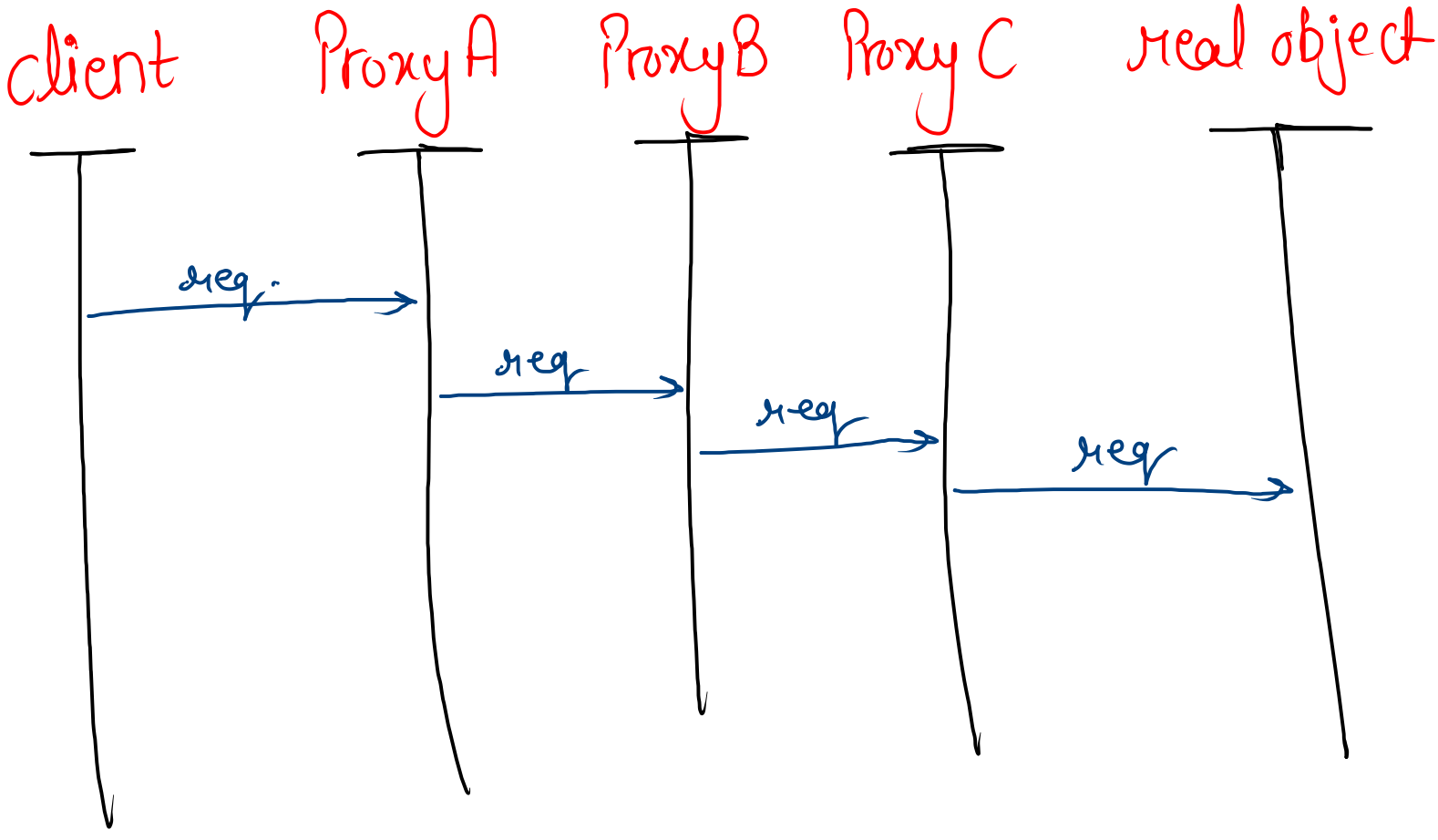
```java
public class EmployeeDaoImpl implements EmployeeDao{

    @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



⇒ LLD of handling NULL (null object design pattern)

Problem :- what will happen when vehicle obj. appear as null.

```
private static void printVehicleDetails(Vehicle vehicle){

        System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());
        System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());
}
```

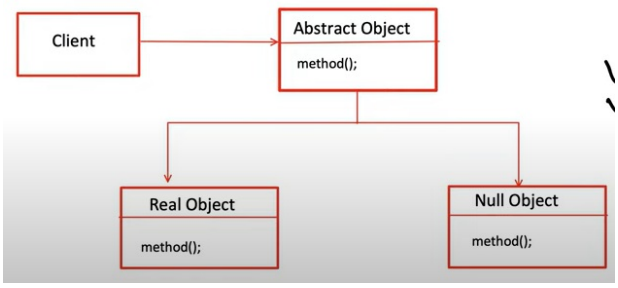
Solution :- But we can't put this check in every single place.

```
private static void printVehicleDetails(Vehicle vehicle) {

    if (vehicle != null) {

        System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());
        System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());
    }
}
```

→ Null object design patter

- → A null object replaces null return type
- → no need to put if check for checking null everytime.
- → null object reflects do nothing or default behaviour.

⇒ UML diagram



client code

```
public class Main {

    public static void main(String args[]){


        Vehicle vehicle = VehicleFactory.getVehicleObject( typeOfVehicle: "Car");
        printVehicleDetails(vehicle);
    }

    private static void printVehicleDetails(Vehicle vehicle) {

            System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());
            System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());
    }
}
```

```
public interface Vehicle {

    int getTankCapacity();
    int getSeatingCapacity();
}
```

```
public class VehicleFactory {


    static Vehicle getVehicleObject(String typeOfVehicle){

        if("Car".equals(typeOfVehicle)) {

            return new Car();
        }

        return new NullVehicle();
    }
}
```

simple factory design

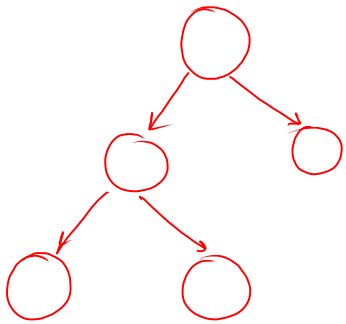
Note:- now we are returning a null object which are reflecting a default behaviour instead of giving error back.

```
public class Car implements Vehicle{

    @Override
    public int getTankCapacity() { return 40; }

    @Override
    public int getSeatingCapacity() { return 5; }
}
```

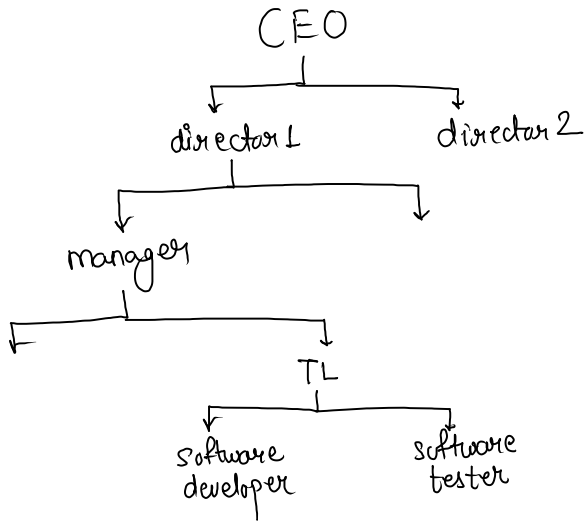
```
public class NullVehicle implements Vehicle{

    @Override
    public int getTankCapacity() { return 0; }

    @Override
    public int getSeatingCapacity() { return 0; }
}
```

⇒ Composite design pattern (Object inside object)

for ex:- a tree structure



⇒



(a tree structure)

Ques we want to create file system

file class

```
public class File {

    String fileName;

    public File(String name) { this.fileName = name; }

    public void ls(){
        System.out.println("file name " + fileName);
    }
}
```

directory class.

```
public class Directory {
    String directoryName;
    List<Object> objectList;

    public Directory(String name){
        this.directoryName = name;
        objectList = new ArrayList<>();
    }

    public void add(Object object) { objectList.add(object); }

    public void ls(){
        System.out.println("Directory Name: " + directoryName);
        for(Object obj: objectList) {

            if(obj instanceof File) {
                ((File) obj).ls();
            }
            else if(obj instanceof Directory) {
                ((Directory) obj).ls();
            }
        }
    }
}
```

Problem:- here we have to create many instance of using if-else, and better way is to use composite design pattern.

file or directory

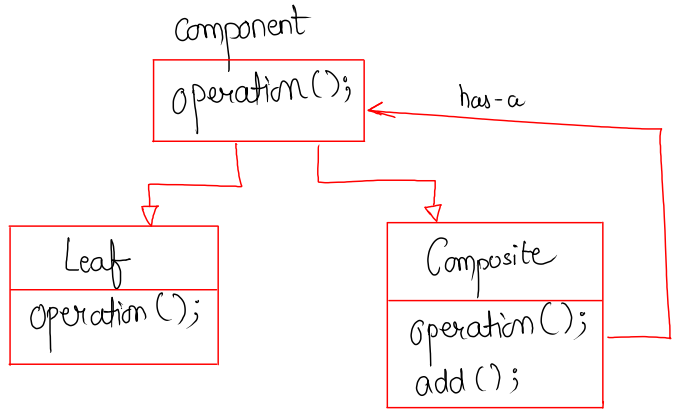
↓

file or directory

↓

file or directory

UML diagram



leaf node of tree

Composite object is which is containg object of itself

client code

```java
public class Main {
    public static void main(String args[]){

        Directory movieDirectory = new Directory( name: "Movie");

         FileSystem border = new File( name: "Border");
         movieDirectory.add(border);

        Directory comedyMovieDirectory = new Directory( name: "ComedyMovie");
        File hulchul = new File( name: "Hulchul");
         comedyMovieDirectory.add(hulchul);
         movieDirectory.add(comedyMovieDirectory);

         movieDirectory.ls();

    }
}
```

Interface

```java
public interface FileSystem {
    public void ls();
}
```

```java
public class File implements FileSystem{
    String fileName;

    public File(String name) { this.fileName = name; }

    public void ls(){
        System.out.println("file name " + fileName);
    }
}
```

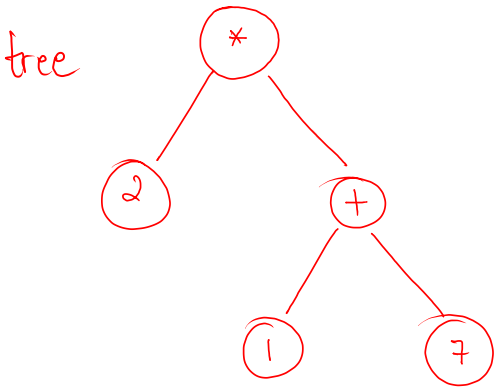
```java
public class Directory implements FileSystem {
    String directoryName;
    List<FileSystem> fileSystemList;

    public Directory(String name){
        this.directoryName = name;
        fileSystemList = new ArrayList<>();
    }

    public void add(FileSystem fileSystemObj) { fileSystemList.add(fileSystemObj); }

    public void ls(){
        System.out.println("Directory name " + directoryName);

        for(FileSystem fileSystemObj : fileSystemList){
            fileSystemObj.ls();
        }
    }
}
```

file system list

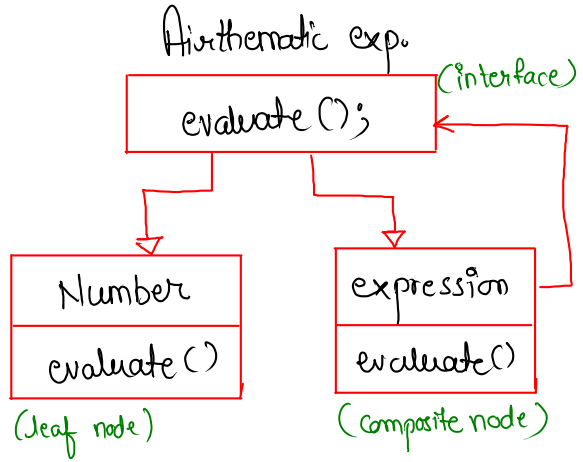
now we don't need to check if-else cond^n here, because ls() funtion is present in the interface itself

Ex:- design a calculator / expression evaluator

↳ 2 * (1 + 7)



leaf node will be a no. and other nodes (composite) will contain left exp and right expression.




```java
public enum Operation {

    ADD,
    SUBTRACT,
    MULTIPLY,
    DIVIDE;
}
```

```java
public interface ArithmeticExpression {

    public int evaluate();
}
```

```java
public class Number implements ArithmeticExpression{
    int value;

    public Number(int value) { this.value = value; }

    public int evaluate(){
        System.out.println("Number value is :" + value);
        return value;
    }
}
```

```java
public class Expression implements ArithmeticExpression {

    ArithmeticExpression leftExpression;
    ArithmeticExpression rightExpression;
    Operation operation;

    public Expression(ArithmeticExpression leftPart, ArithmeticExpression rightPart, Operation operation){
        this.leftExpression = leftPart;
        this.rightExpression = rightPart;
        this.operation = operation;
    }

    public int evaluate(){

        int value = 0;
        switch (operation){

            case ADD:
                value = leftExpression.evaluate() + rightExpression.evaluate();
                break;
            case SUBTRACT:
                value = leftExpression.evaluate() - rightExpression.evaluate();
                break;
            case DIVIDE:
                value = leftExpression.evaluate() / rightExpression.evaluate();
                break;
            case MULTIPLY:
                value = leftExpression.evaluate() * rightExpression.evaluate();
                break;
        }

        System.out.println("Expression value is :" + value);
        return value;
    }
}
```

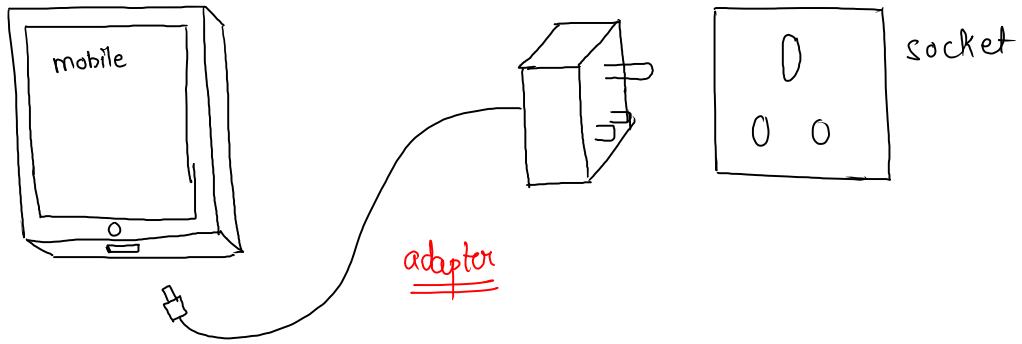
```java
ArithmeticExpression two = new Number( value: 2);

ArithmeticExpression one = new Number( value: 1);
ArithmeticExpression seven = new Number( value: 7);
ArithmeticExpression addExpression = new Expression(one,seven, Operation.ADD);

ArithmeticExpression parentExpression = new Expression(two,addExpression, Operation.MULTIPLY);

System.out.println(parentExpression.evaluate());
```

Client code, here we are creating an expression.

⇒ Adapter design pattern

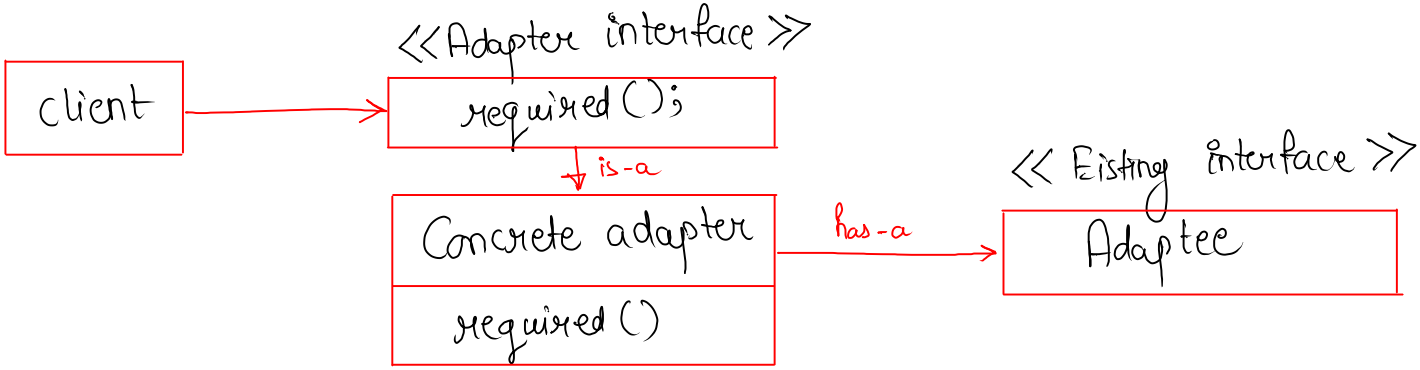
It is a bridge b/w existing interface & expected interface

Ex1

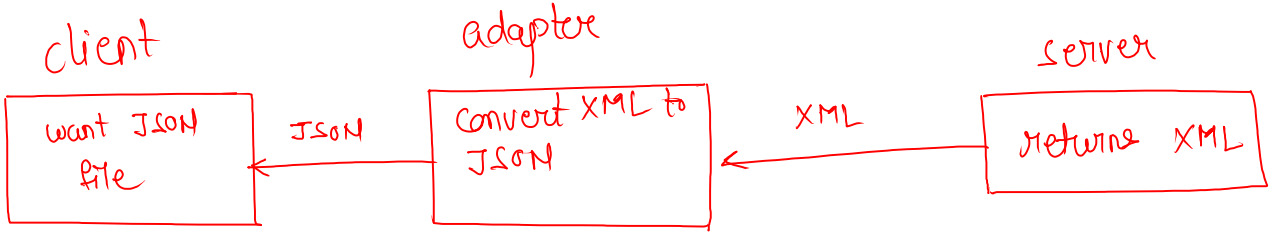


(here mobile is not compatible to directly get charging from socket, so we need a adapter)

This structure is used when we want to make 2 unrelated interfaces work together. It is often used to make existing class work with other without modifying the source code.



Ex2 XML to JSON parser



Ques design a weighting machine

(client might need weight in KGs but machine is returning it in pounds, so we need to create an adapter)

client code

```
public class Main {
    public static void main(String args[]){
        WeightMachineAdapter weightMachineAdapter = new WeightMachineAdapterImpl(new WeightMachineForBabies());
        System.out.println(weightMachineAdapter.getWeightInKg());
    }
}
```

adapter

```
public interface WeightMachineAdapter {
    public double getWeightInKg();
}
```

client wanted in kg

```
public class WeightMachineAdapterImpl implements WeightMachineAdapter{
    WeightMachine weightMachine;
    public WeightMachineAdapterImpl(WeightMachine weightMachine) { this.weightMachine = weightMachine; }
    @Override
    public double getWeightInKg() {
        double weightInPound = weightMachine.getWeightInPound();
        //Convert it to KGs
        double weightInKg = weightInPound * .45;
        return weightInKg;
    }
}
```

adaptee

```
public interface WeightMachine {
    //return the weight in Pound
    public double getWeightInPound();
}
```

```
public class WeightMachineForBabies implements WeightMachine{
    @Override
    public double getWeightInPound() { return 28; }
}
```

existing interface

⇒ Builder design pattern

This pattern is used when dealing with objects that have large no. of optional parameters or configuration.

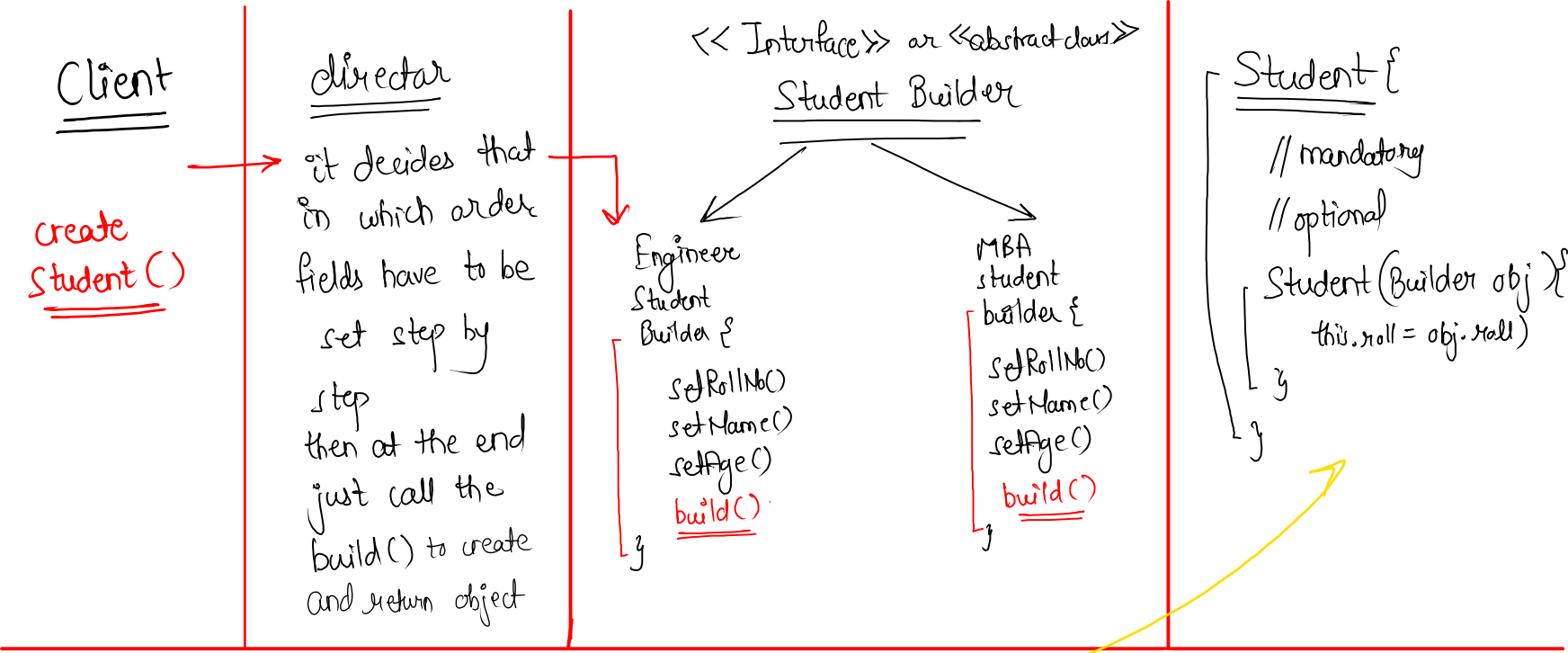
Problem

```
no usages
public class Student {
    1 usage
    int rollNumber;
    1 usage
    int age;
    1 usage
    String name;
    1 usage
    String fatherName;
    1 usage
    String motherName;
    1 usage
    List<String> subjects;
    1 usage
    String mobileNumber;


    no usages
    public Student(int rollNumber, int age, String name, String fatherName, String motherName, List<String> subjects, String mobileNumber){
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.fatherName = fatherName;
        this.motherName = motherName;
        this.subjects = subjects;
        this.mobileNumber = mobileNumber;
    }
}
```

→ only mandatory and all other are optional

Now, we need to have lots of constructors

So builder design pattern used to create objects step by step.



Note:- In student class, now dont need to have many constructor also we don't need to pass many fields in constructor.
We are just passing 1 parameter of type StudentBuilder

Note:- Also return type of every method in StudentBuilder class is StudentBuilder (which is a mediator object) except build method which has a return type Student that is why we are definiately calling it at the end

Code

Director

```
public class Director {

    StudentBuilder studentBuilder;

    Director(StudentBuilder studentBuilder){
        this.studentBuilder = studentBuilder;
    }

    public Student createStudent(){

        if(studentBuilder instanceof EngineeringStudentBuilder){
            return createEngineeringStudent();
        }
        else if(studentBuilder instanceof MBAStudentBuilder){
            return createMBAStudent();
        }
        return null;
    }

    private Student createEngineeringStudent(){

        return studentBuilder.setRollNumber(1).setAge(22).setName("sj").setSubjects().build();
    }

    private Student createMBAStudent(){

        return studentBuilder.setRollNumber(2).setAge(24).setName("sj").setFatherName("MyFatherName").setMotherName("MyMotherName").setSubjects().build();
    }
}
```

creating each field step by step (not necessarily all) and then at the end call build() to return a Student type of object.

StudentBuilder

```
public abstract class StudentBuilder {

    int rollNumber;
    int age;
    String name;
    String fatherName;
    String motherName;
    List<String> subjects;

    public StudentBuilder setRollNumber(int rollNumber) {
        this.rollNumber = rollNumber;
        return this;
    }

    public StudentBuilder setAge(int age) {
        this.age = age;
        return this;
    }

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

    public StudentBuilder setFatherName(String fatherName) {
        this.fatherName = fatherName;
        return this;
    }

    public StudentBuilder setMotherName(String motherName) {
        this.motherName = motherName;
        return this;
    }

    abstract public StudentBuilder setSubjects();

    public Student build() {
        return new Student(this);
    }
}
```

return type is student

Student

```
public class Student {
    int rollNumber;
    int age;
    String name;
    String fatherName;
    String motherName;
    List<String> subjects;

    public Student(StudentBuilder builder){
        this.rollNumber = builder.rollNumber;
        this.age = builder.age;
        this.name = builder.name;
        this.fatherName = builder.fatherName;
        this.motherName = builder.motherName;
        this.subjects = builder.subjects;
    }
}
```

only 1 para. needed

```
public class Client {

    public static void main(String args[]){

        Director directorObj1 = new Director(new EngineeringStudentBuilder());
        Director directorObj2 = new Director(new MBAStudentBuilder());

        Student engineerStudent = directorObj1.createStudent();
        Student mbaStudent =directorObj2.createStudent();

        System.out.println(engineerStudent.toString());
        System.out.println(mbaStudent.toString());

    }
}
```

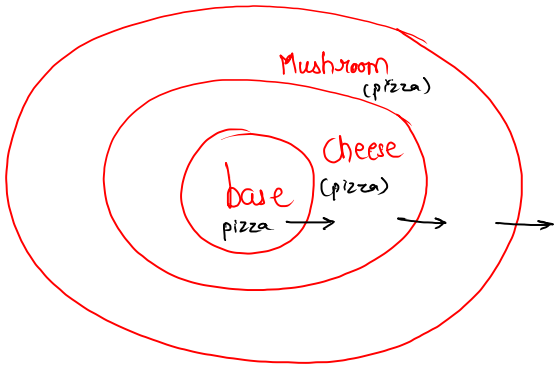
client code become easy

```
public class MBAStudentBuilder extends StudentBuilder{

    @Override
    public StudentBuilder setSubjects() {
        List<String> subs = new ArrayList<>();
        subs.add("Micro Economics");
        subs.add("Business Studies");
        subs.add("Operations Management");
        this.subjects = subs;
        return this;
    }
}
```

```
public class EngineeringStudentBuilder extends StudentBuilder{

    @Override
    public StudentBuilder setSubjects() {
        List<String> subs = new ArrayList<>();
        subs.add("DSA");
        subs.add("OS");
        subs.add("Computer Architecture");
        this.subjects = subs;
        return this;
    }
}
```

Ques What is the difference b/w builder & decorator design pattern with respect to pizza problem.

decorator



decorator is used to add additional attributes of an existing obj. dynamically to create a new obj.
Unlike builder, there are no restriction of finalizing the obj until all its attributes are added.

Client →

Director

request for → Base Pizza + Cheese

it will call first class and call it's steps

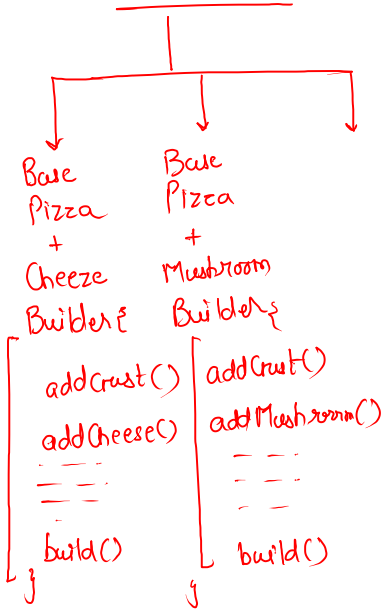
request for → Base Pizza + Mushroom

it will call second class and call it's steps

request for → Base Pizza + Cheese + Mush room

not possible bcz we do not have a class like that bcz builder cannot handle dynamic request

Pizza Builder

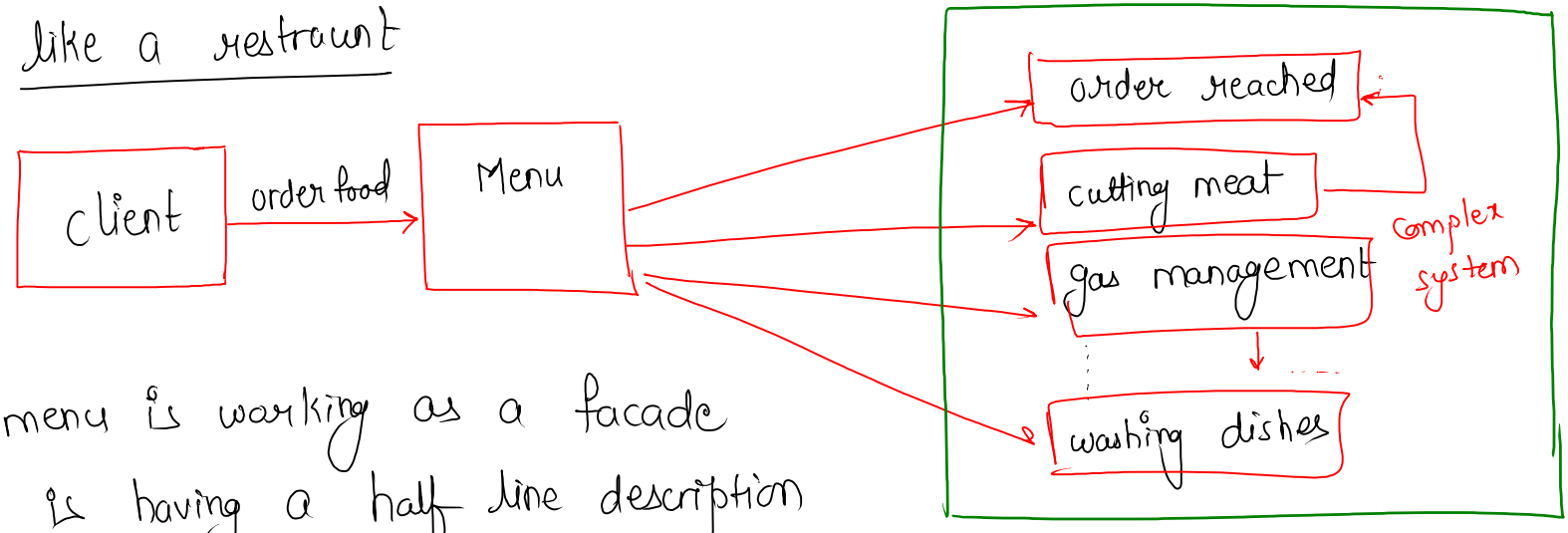



```
Pizza {
    Base base;
    Cheese cheese;
    Mushroom mushroom;
    ---
    ---
    ---
}
```

⇒ Facade design pattern

↳ Widely used when we have to hide the system complexity from client

Ex:- like a restraunt



here, menu is working as a facade which is having a half line description of dish and client doesn't need to know that how the actual complex system is working inside kitchen.