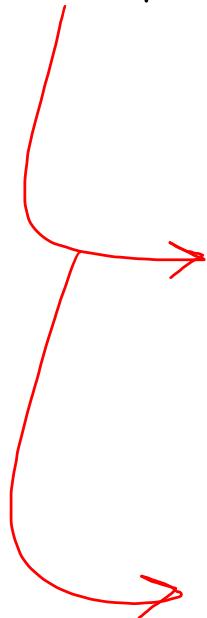


Pre - requisite



OOP's Concepts

A red hand-drawn arrow pointing from the 'OOP's Concepts' text down towards the 'SOLID Principles' text.

SOLID Principles

→ Design Patterns

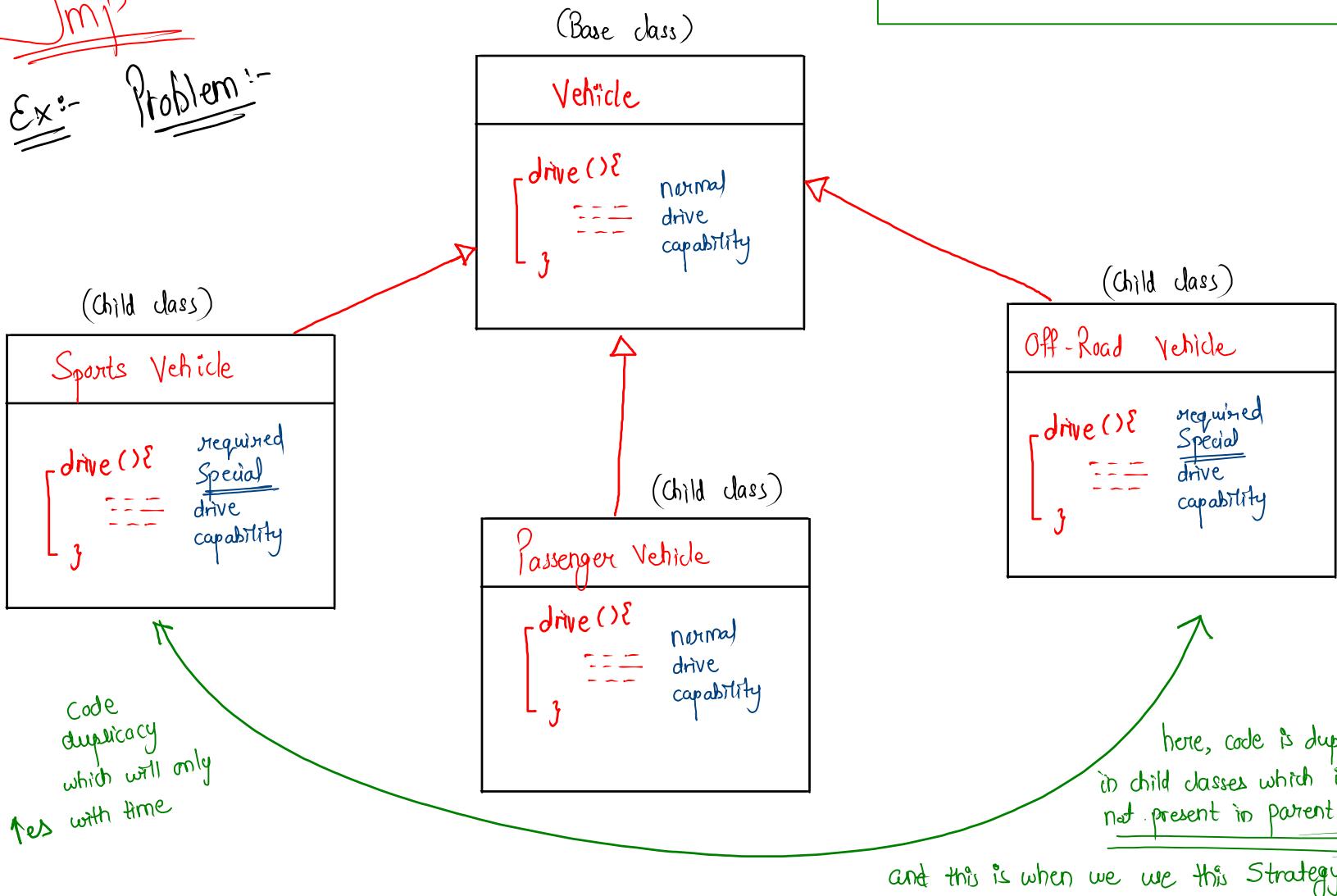
design patterns are solution to common problems encountered by software developers when designing and building applications.

- 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

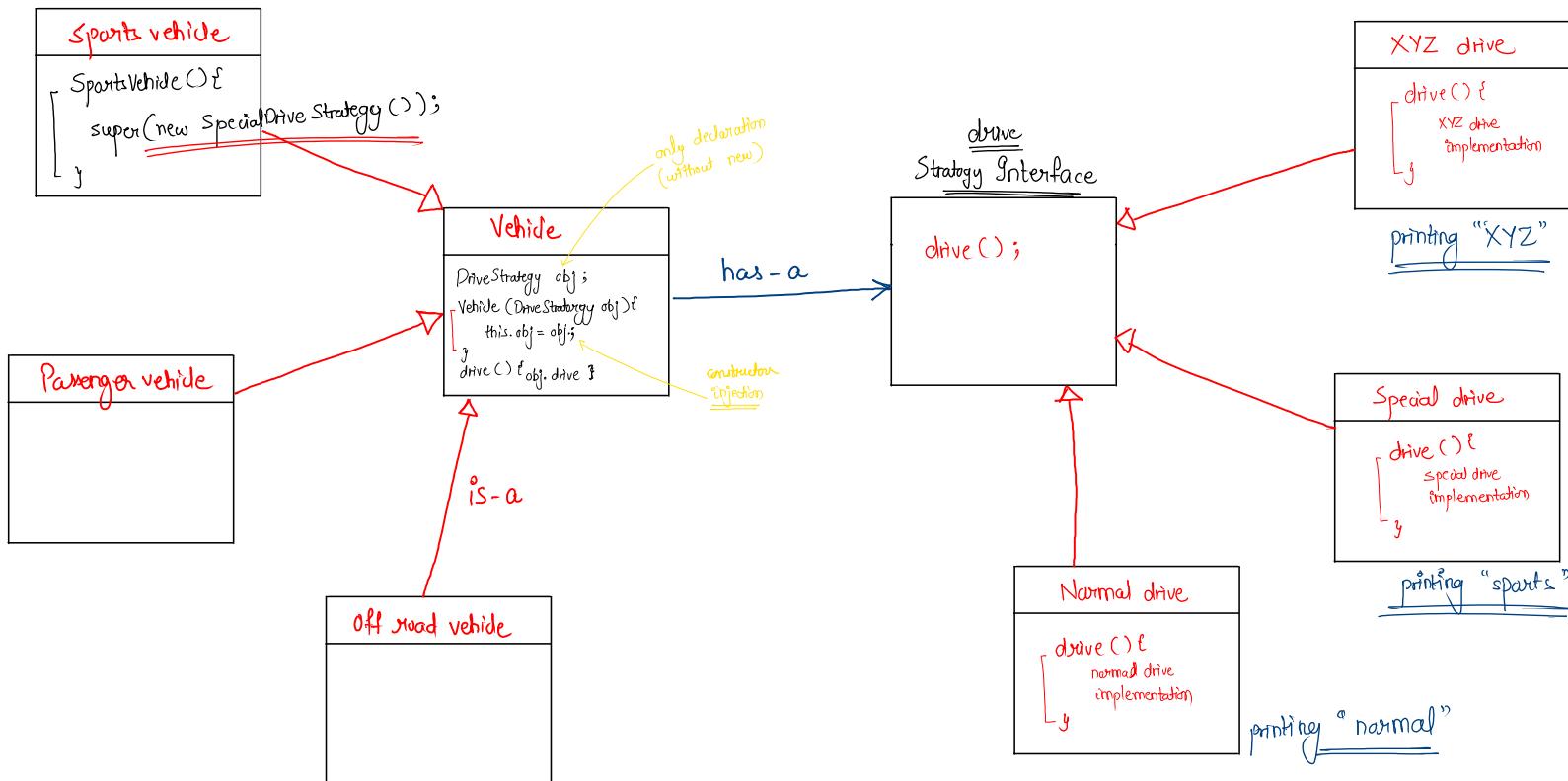
~~JMP~~

Ex:- Problem :-



Solution :-

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



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

Client code

```
public class Main {  
    public static void main(String[] args) {  
        Vehide obj = new SportsVehide();  
        obj.drive(); // print sports  
  
        Vehide obj = new NormalDrive();  
        obj.drive(); // print normal  
    }  
}
```

→ Observer Design Pattern :- (Walmart interview) question

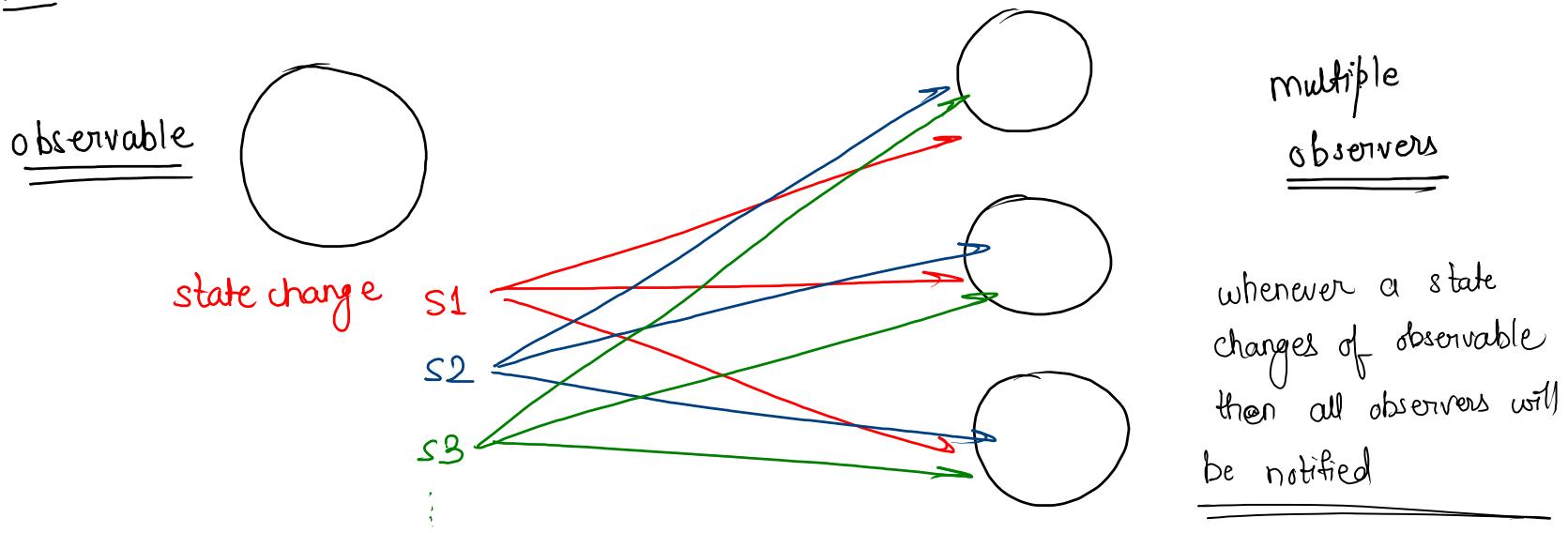
Jue)

amazon.com
product is unavailable
notify me button

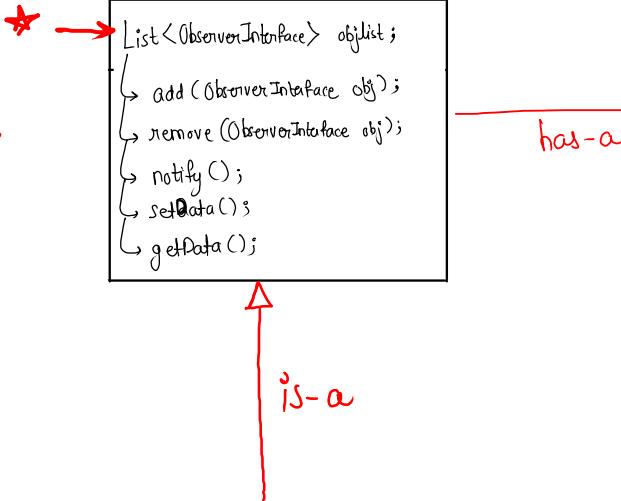
In amazon.com, we are looking for a product which is unavailable and there is a button of notify me?, so send notifications to customers when product is available.

Implement this button (LLD questions)

Note:- There are 2 states



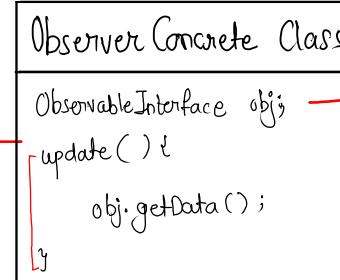
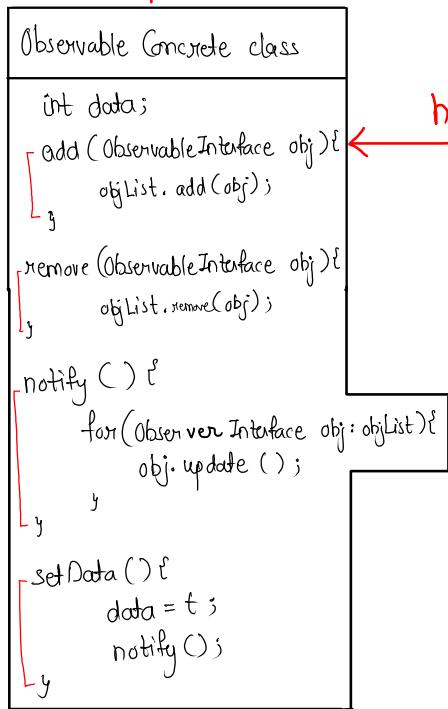
Observable Interface



Observer Interface

```
update();
```

there can
be multiple
concrete
classes



here this obj is used
to know that which
concrete class we are
referencing 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 tempo every hour

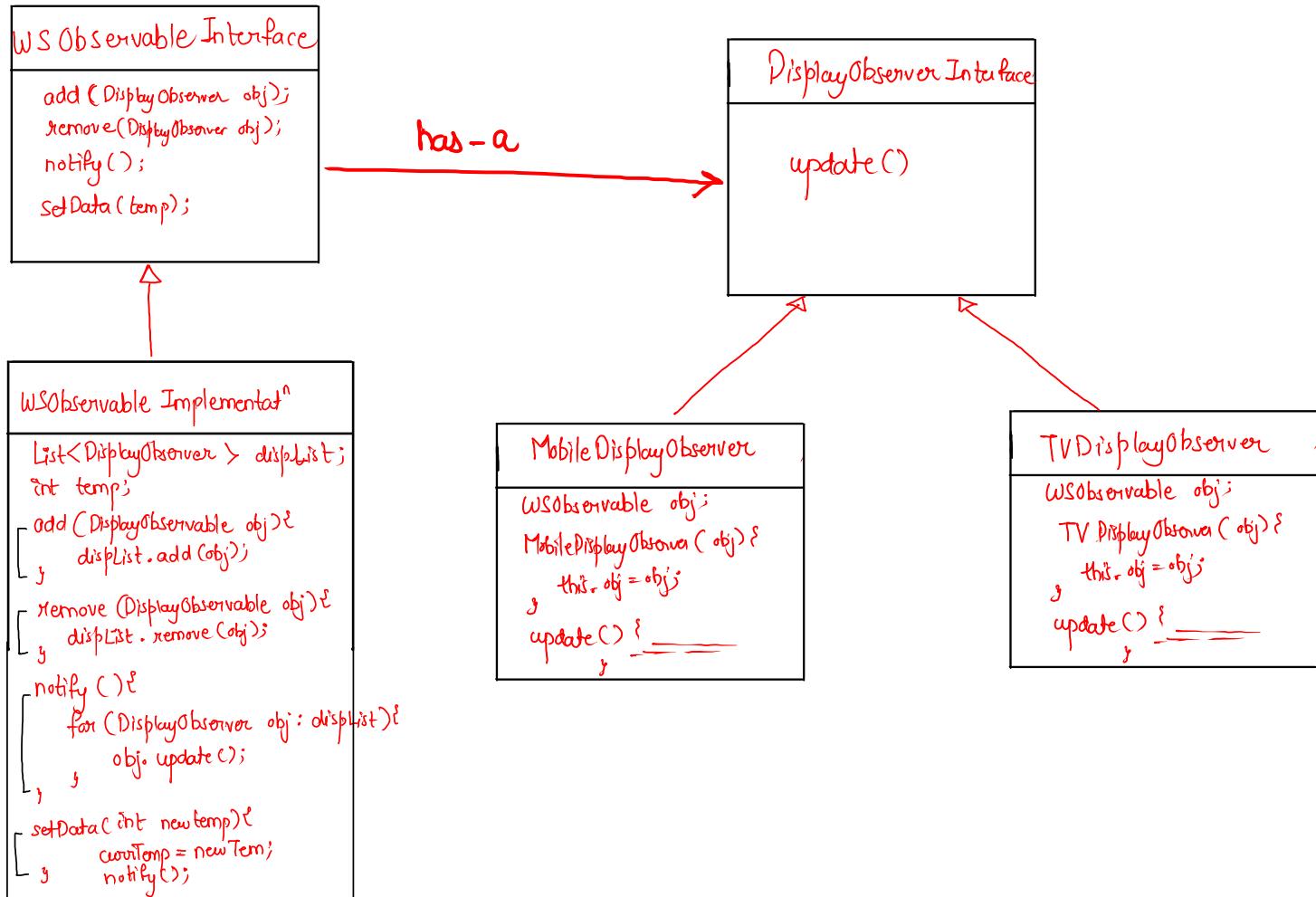
observed by

TV display observer
X

Mobile display observer

Solution

Note :- WS Observable = weather station observable



Solution of Walmart Interview Question :-

```
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

```
public interface NotificationAlertObserver {  
    public void update();  
}
```

```
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; }  
}
```

```
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, "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  
    }  
}
```

```
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, "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*

Decorator Design Pattern

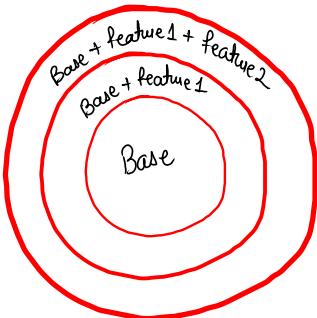
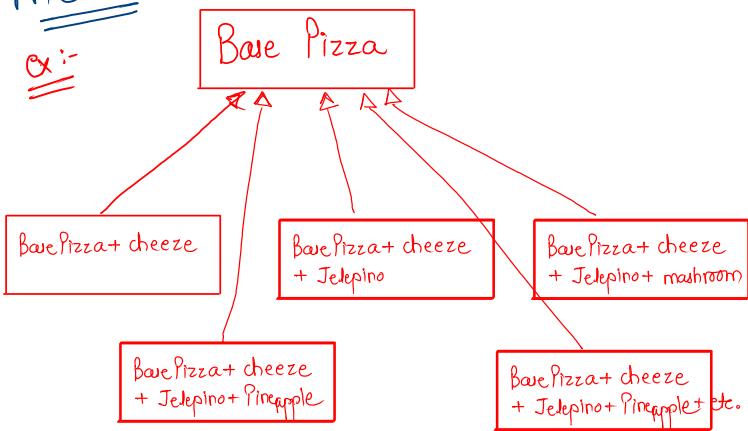
Ex:- Coffee machine design, pizza design,
etc In question

Why do we need decorator pattern?

To avoid class explosion

means:-

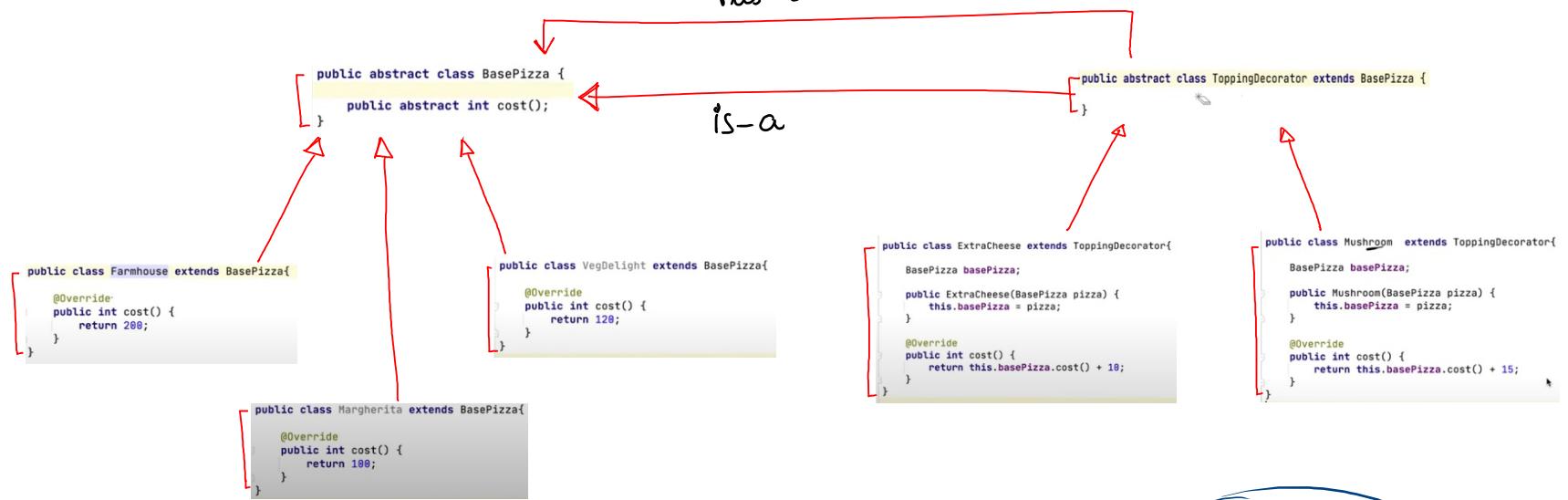
Ex:-



This is where decorator 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.

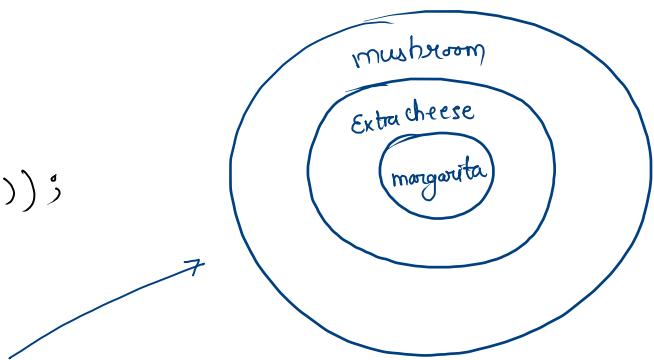
Like, that how many different classes are we going to make with different combination, so it will be very difficult to manage.

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



Client code

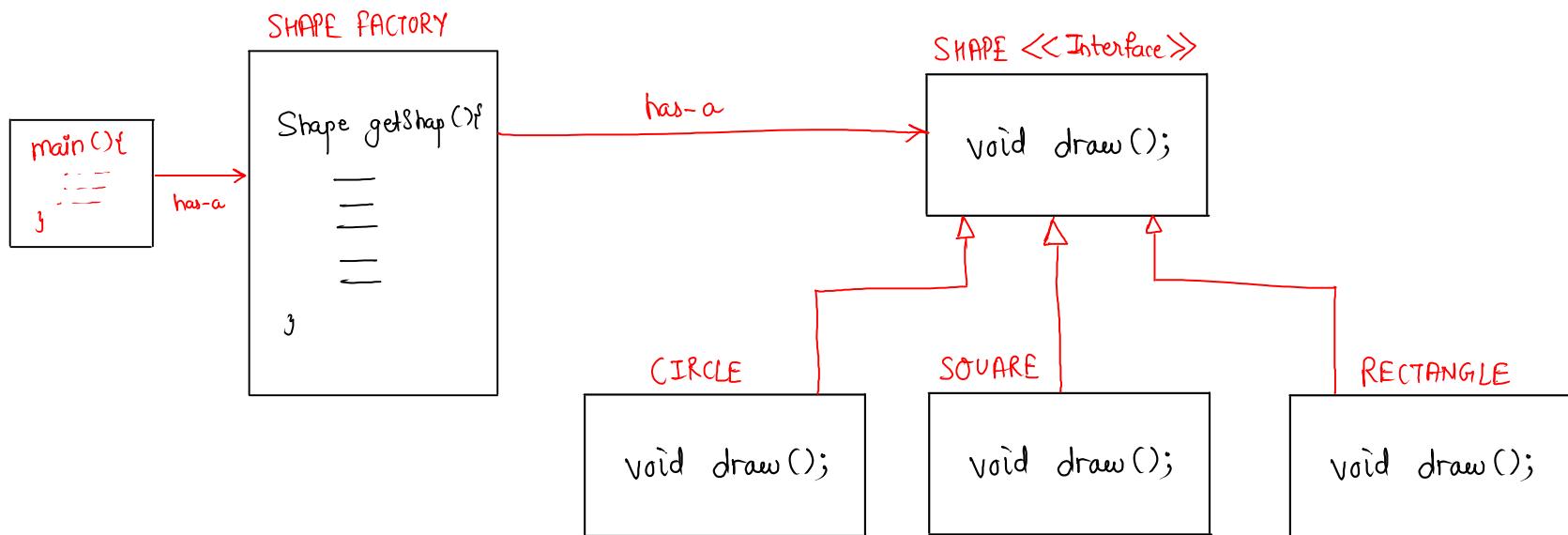
↳ Pizza pizza = new ExtraCheese(new Margarita());
 pizza.cost() // $100 + 10 = 110$



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

⇒ Factory Pattern ~~V. gmp~~

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



Example code

```
public class MainClass {  
    public static void main(String args[]) {  
        ShapeFactory shapeFactoryObj = new ShapeFactory();  
        Shape shapeObj = shapeFactoryObj.getShape( input: "CIRCLE" );  
        shapeObj.draw();  
    }  
}
```



```
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 {  
    void draw();  
}
```

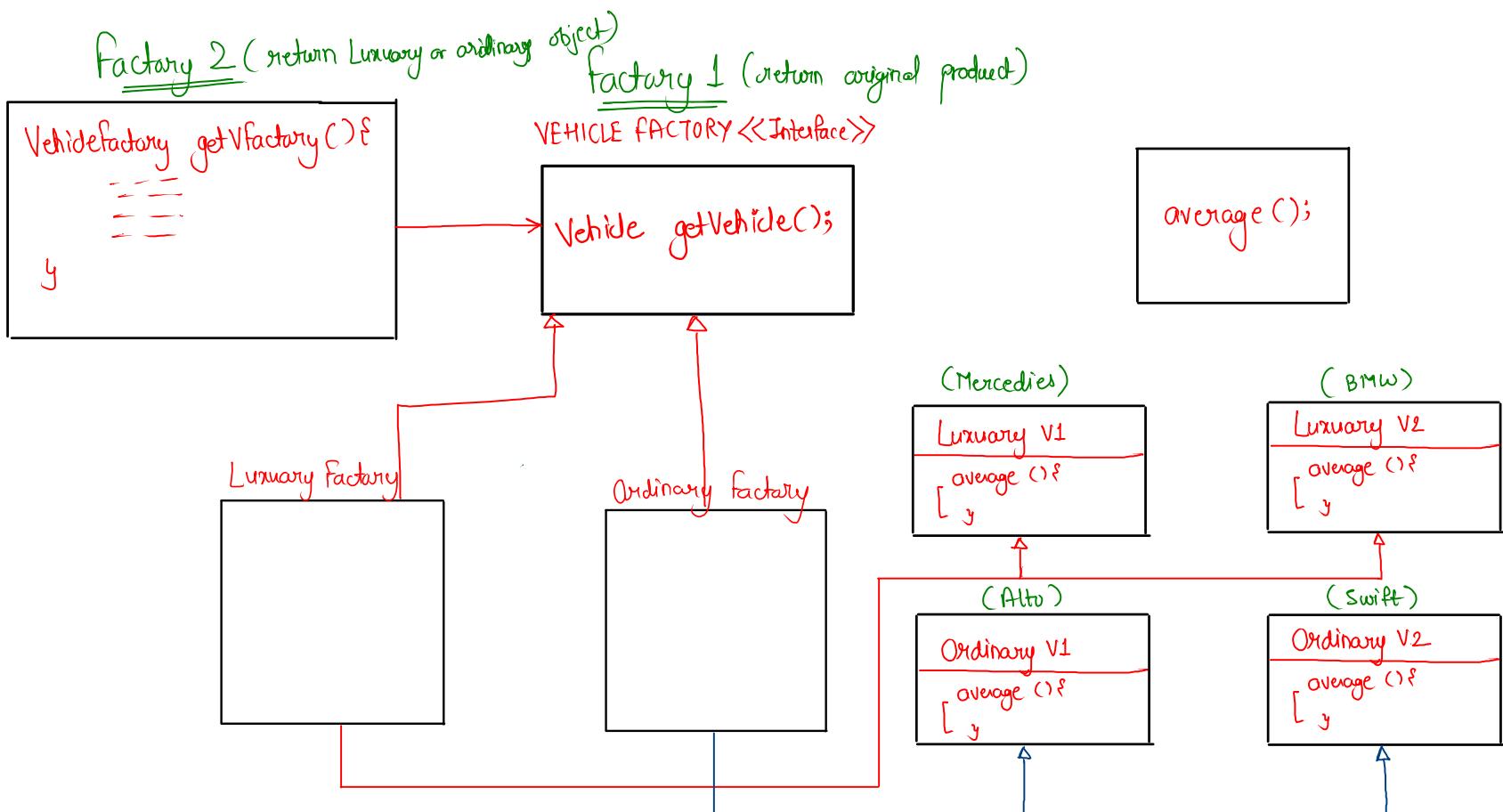
```
public class Rectangle implements Shape{  
    @Override  
    public void draw() {  
        System.out.println("rectangle");  
    }  
}
```

```
public class Circle implements Shape{  
    @Override  
    public void draw() {  
        System.out.println("circle");  
    }  
}
```

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 separately.



Ques Design Tic-Tac-Toe Game

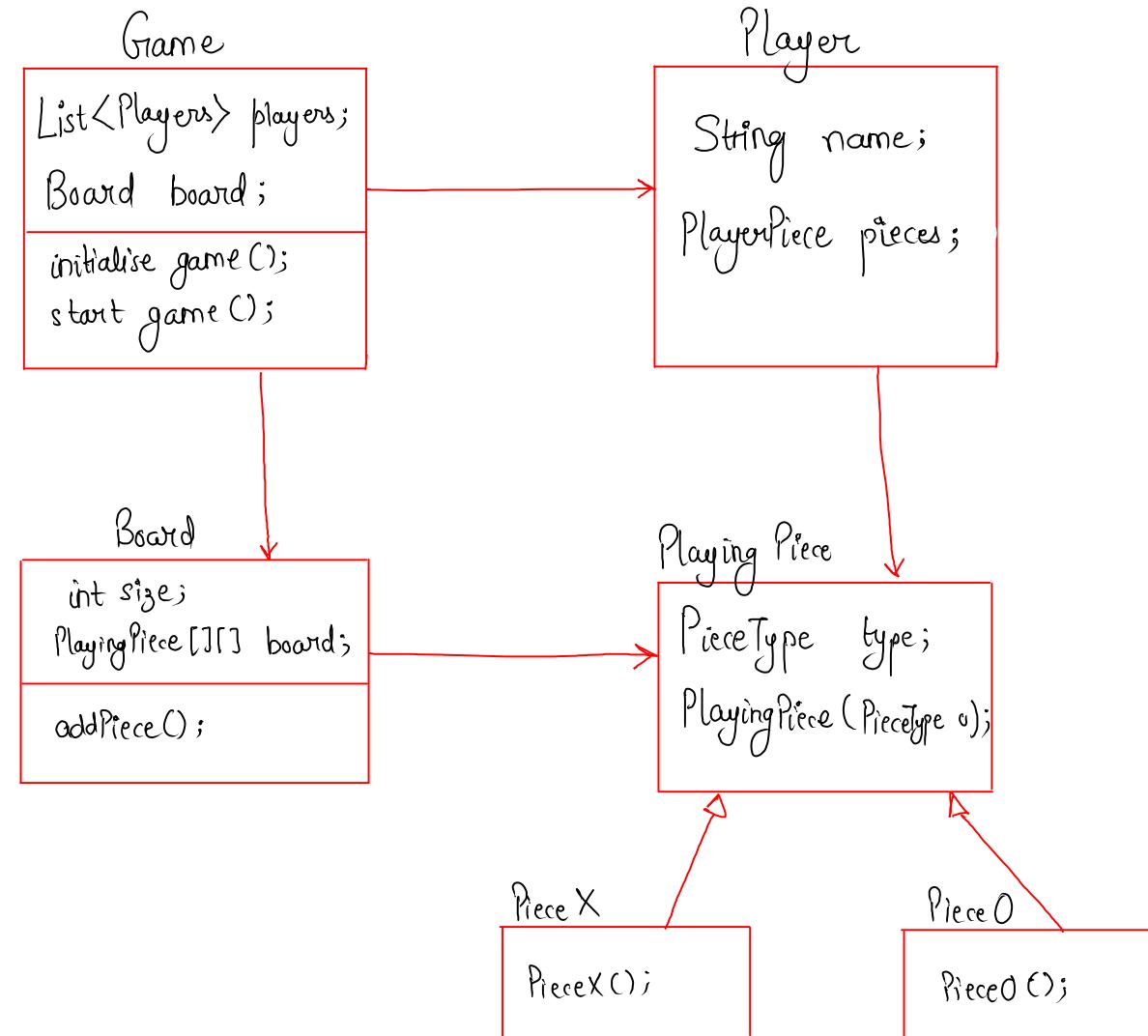
UML Diagram :-

enum PieceType {

X,

O;

y



~~Complete code~~

Game

```
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();  
    }  
}
```

```
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;  
    }  
}
```

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

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

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

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

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

    public boolean isThereAWinner(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());
    }
}

```

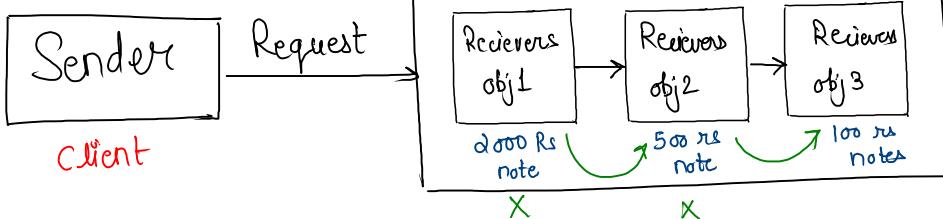
⇒ Chain of responsibility Design Pattern

~~Application usage :-~~

- ATM / Vending machine
- Design logger (Amazon)

structure

Ex:- you (withdraw 2000)

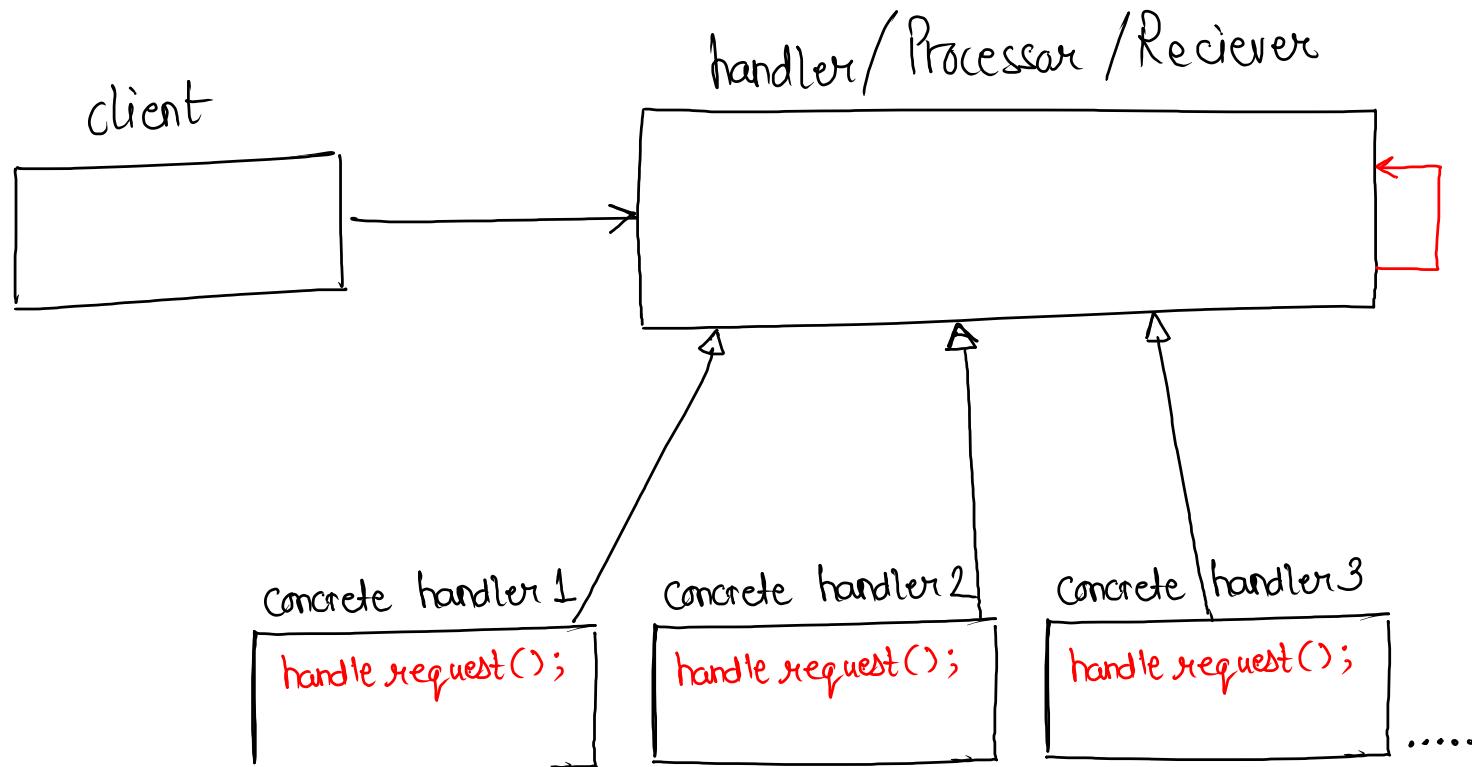


Note:- This type of design is used when we client sends a request and it doesn't matter that who is completing that request

Ex:-

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

Structure



One Design logger

```
Logger obj = new Logger();
```

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

Code

```
public class Main {  
    public static void main(String args[]) {  
        LogProcessor logObject = new InfoLogProcessor(new DebugLogProcessor(new ErrorLogProcessor(nextLogProcessor: 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 ");  
    }  
}
```

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

here this chaining is imp

```
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);  
        }  
    }  
}
```

here constructor is already storing
next logger processor

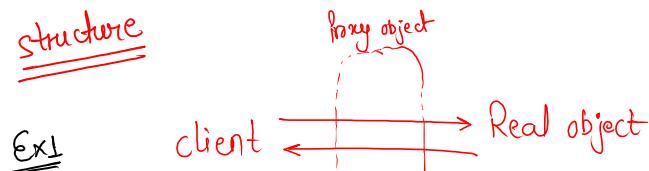
```
public class InfoLogProcessor extends LogProcessor{  
    InfoLogProcessor(LogProcessor nexLogProcessor){  
        super(nexLogProcessor);  
    }  
  
    public void log(int logLevel, String message){  
        if(logLevel == INFO) {  
            System.out.println("INFO: " + message);  
   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

```
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);  
        }  
    }  
}
```

```
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)

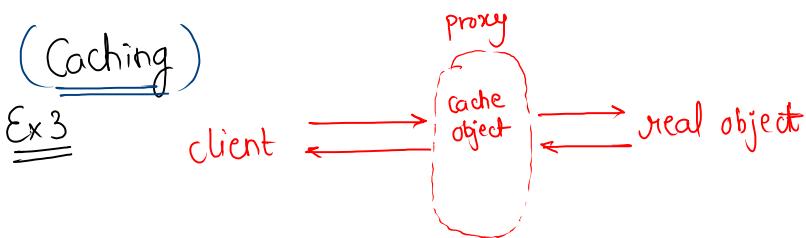
structure



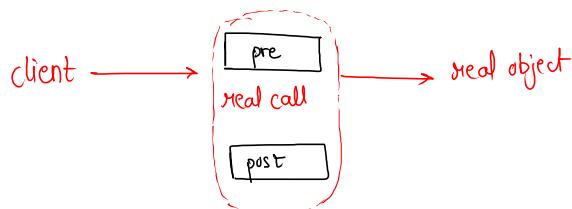
(Internet Restriction)



(Caching)



(Preprocessing & postprocessing)



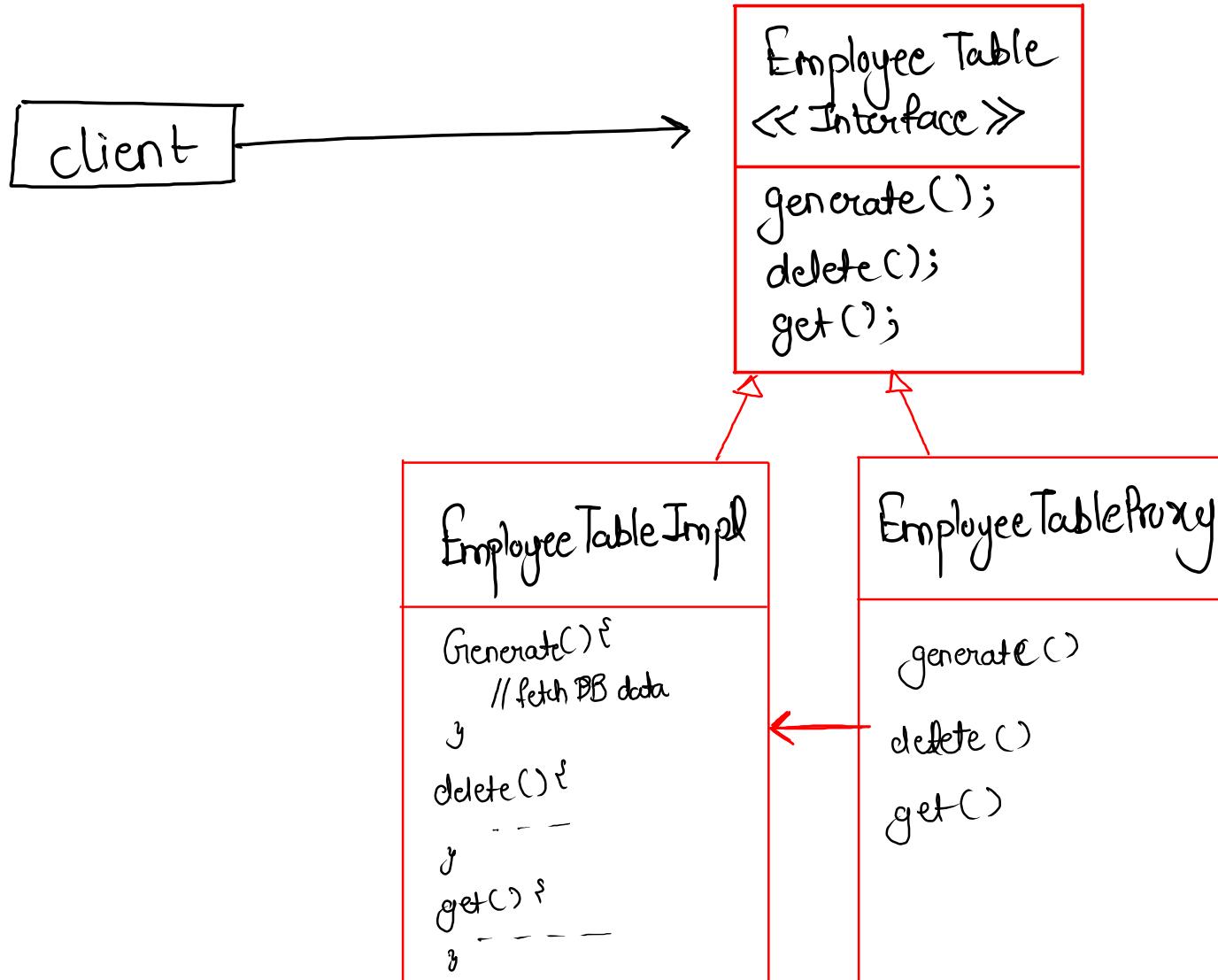
If a client wants to access the real object then proxy object is always inbetween and access the request before approving it.

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

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.

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

structure



Code

```
public class ProxyDesignPattern {  
    public static void main(String args[]) {  
        try {  
            EmployeeDao empTableObj = new EmployeeDaoProxy();  
            empTableObj.create("USER", new EmployeeDo());  
            System.out.println("Operation successful");  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

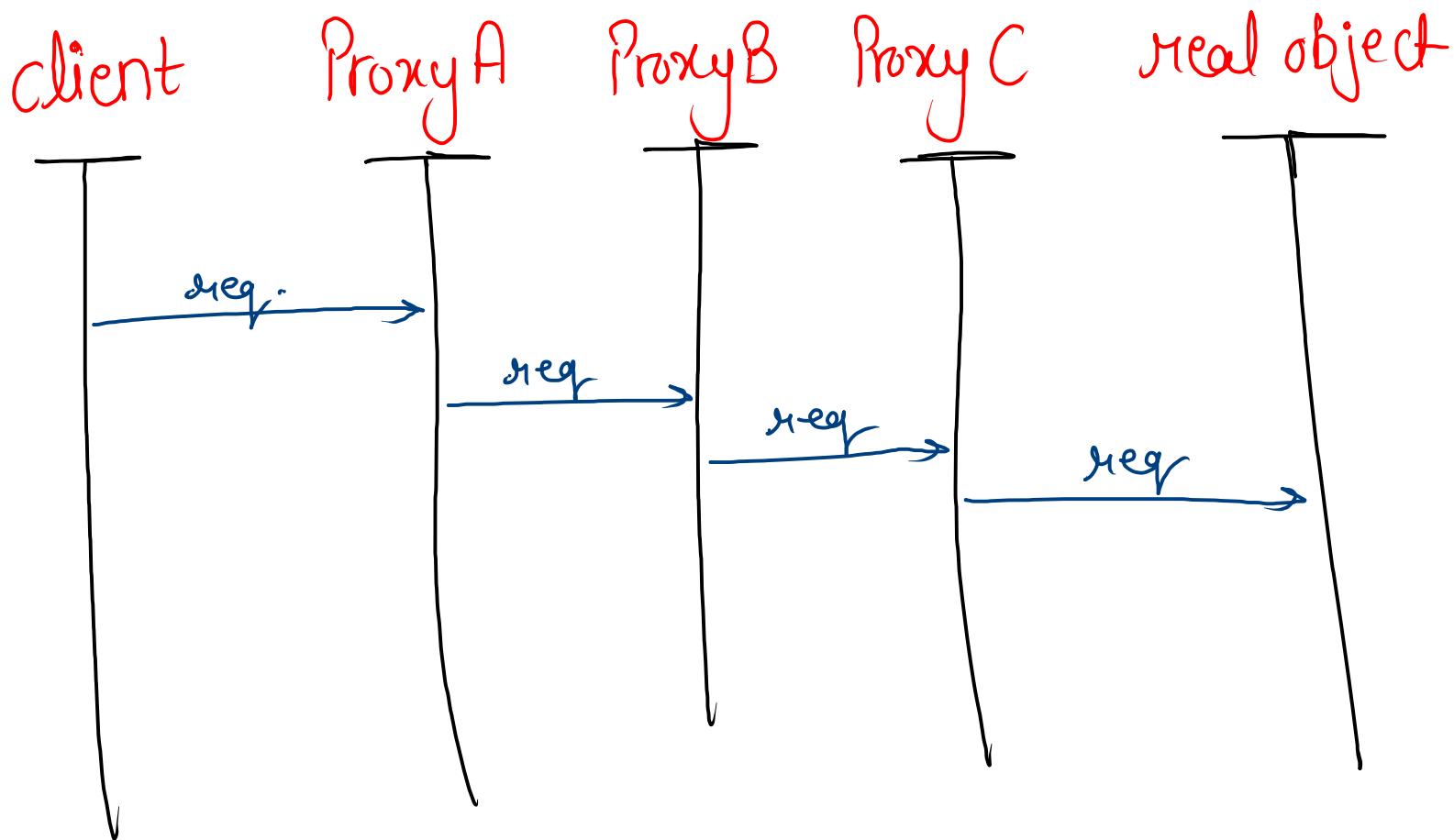
```
public interface EmployeeDao {  
    public void create(String client, EmployeeDo obj) throws Exception;  
    public void delete(String client, int employeeId) throws Exception;  
    public EmployeeDo get(String client, int employeeId) throws Exception;  
}
```

```
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 employeeID:" + employeeId);  
    }  
    @Override  
    public EmployeeDo get(String client, int employeeId) throws Exception {  
        //fetch row  
        System.out.println("fetching data from the DB");  
        return new EmployeeDo();  
    }  
}
```

Gmp

```
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);  
        }  
    }  
}
```

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.

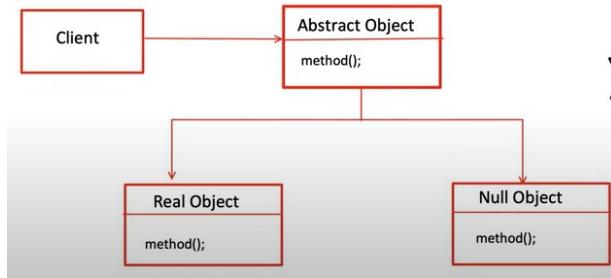
```
private static void printVehicleDetails(Vehicle vehicle){  
    System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());  
    System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());  
}
```

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 pattern
 - 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());
    }
}
  
```

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

```

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

```

public class Car implements Vehicle{
    @Override
    public int getTankCapacity() { return 40; }

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

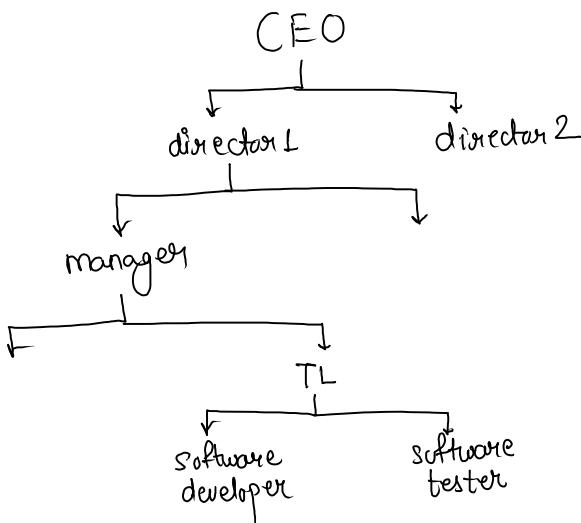
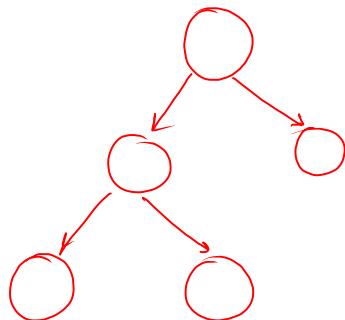
```

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);  
    }  
}
```

problem:- here we have to create many instance of writing if-else, and better way is to use Composite design pattern.

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();  
            }  
        }  
    }  
}
```

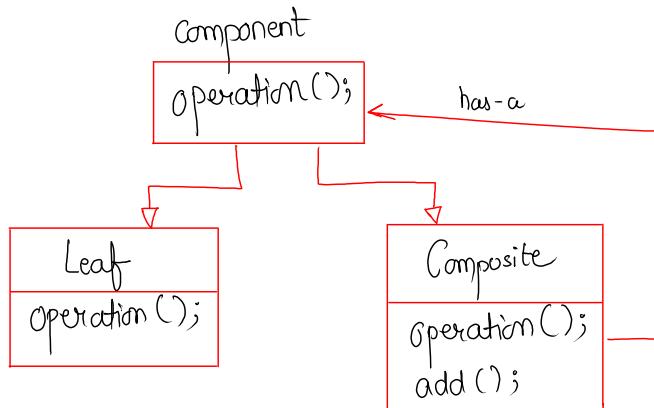
file or directory

file or directory

file or directory

UML diagram

leaf node
of tree



Composite object is which
is containing object of itself

client code

```

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

```

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

```

public class File implements FileSystem{
    String fileName;

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

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

```

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();
        }
    }
}
  
```

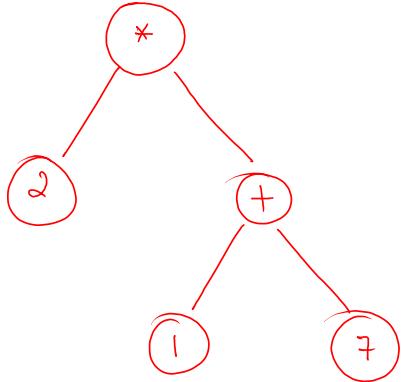
file system list

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

Ex:- design a calculator/ expression evaluator

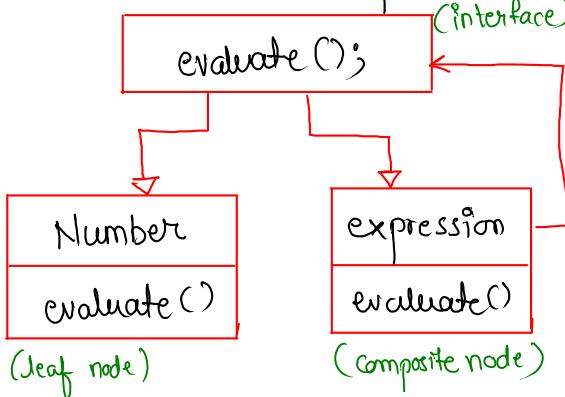
↳ $2 * (1 + 7)$

tree



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

Arithmetic exp.



```

public enum Operation {
    ADD,
    SUBTRACT,
    MULTIPLY,
    DIVIDE;
}
  
```

```

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;
    }
}
  
```

```

public interface ArithmeticExpression {
    public int evaluate();
}
  
```

```

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.

```

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){

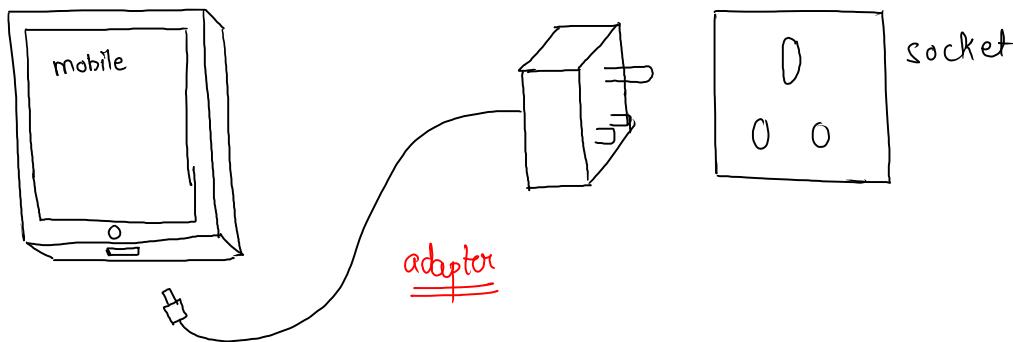
            case ADD:
                value = leftExpression.evaluate() + rightExpression.evaluate();
                break;
            case SUBTRACT:
                value = leftExpression.evaluate() - rightExpression.evaluate();
                break;
            case DIVIDE:
                value = leftExpression.evaluate()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

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

⇒ Adapter design pattern

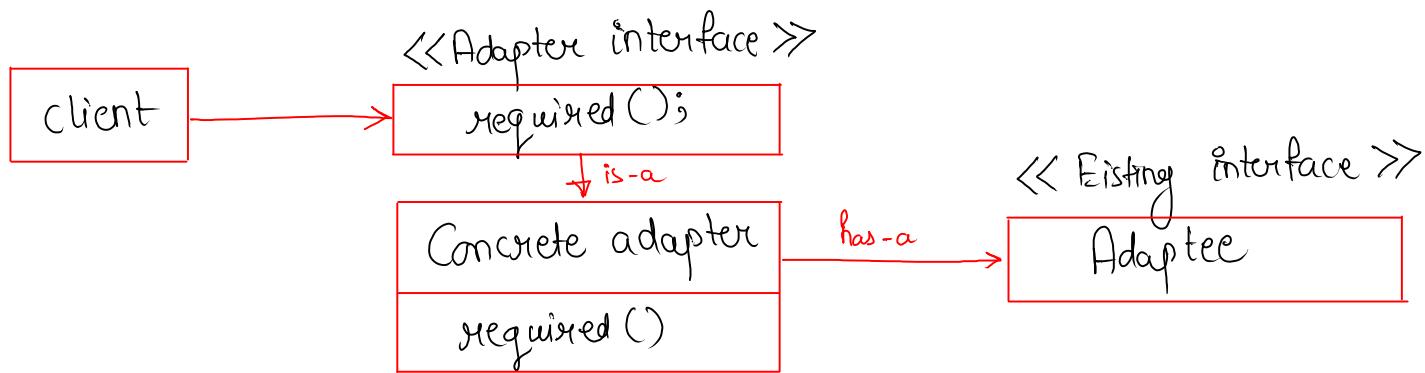
It is a bridge b/w existing interface & expected interface

Ex1



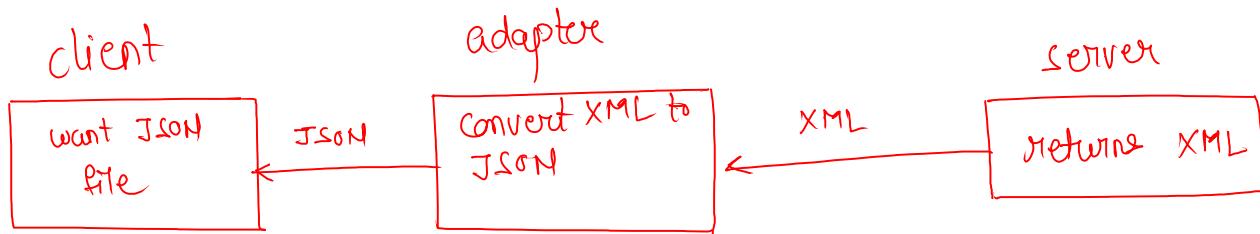
here mobile is not compatible to directly get charging from socket, so we need a adapter

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.



Ex 2

XML to JSON parser



Ques

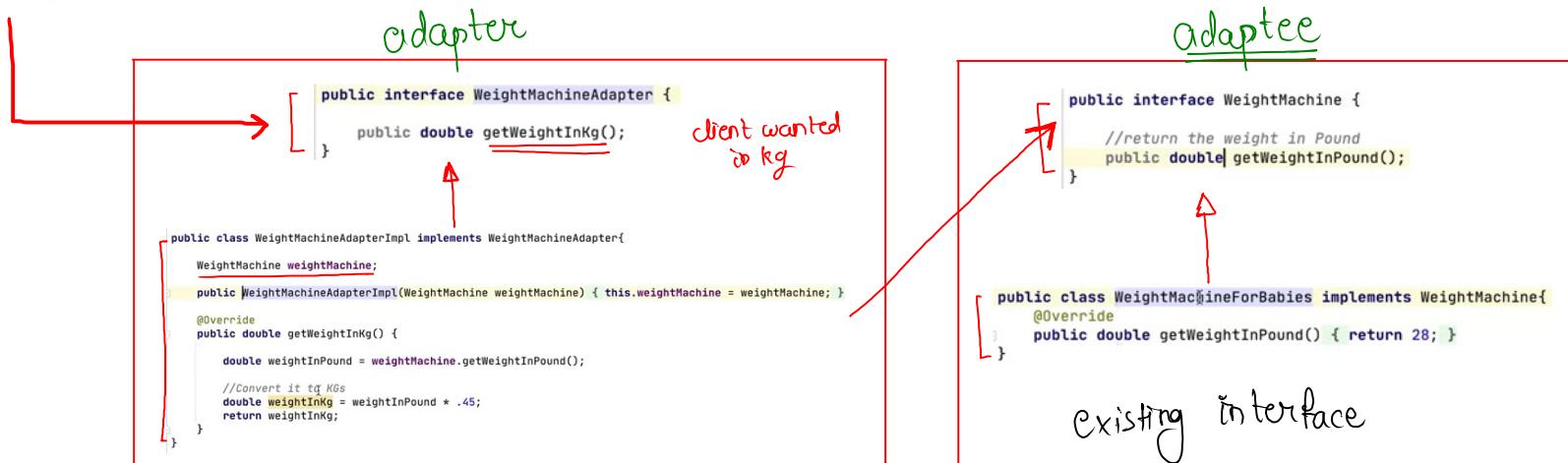
design a weighting machine

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

```

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

client code



→ 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.

Client

create Student()

director

it decides that in which order fields have to be set step by step then at the end just call the build() to create and return object

<< Interface >> or << abstract class >>

Student Builder

Engineer
Student
Builder {

setRollNo()
setName()
setAge()
build()

MBA
student
builder {

setRollNo()
setName()
setAge()
build()

Student {

// mandatory
// optional

Student (Builder obj)
this.roll = obj.roll)

Note :-

In student class, now don't 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 definitely 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.

```

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

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);
    }
}

```

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

return type is student

```

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;
    }
}

```

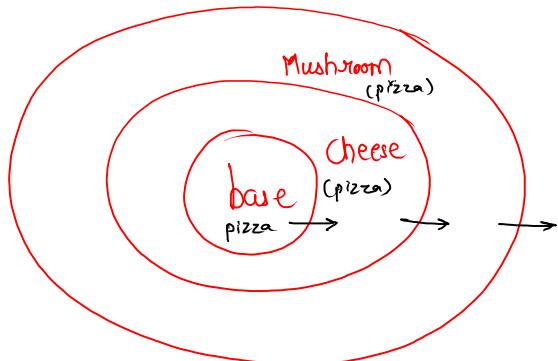
```

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

request for
Base Pizza + Cheese

it will call first class and call
its steps

request for
BasePizza + Mushroom

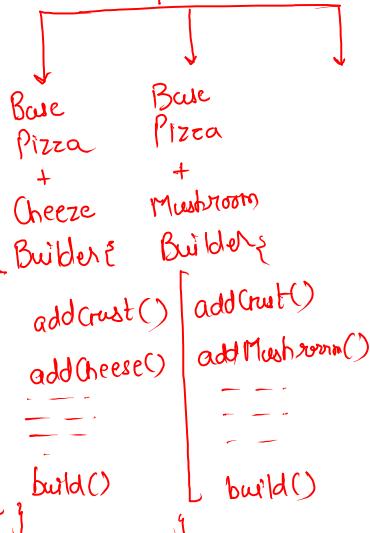
it will call second class and call
its steps

request for
BasePizza + Cheese + Mushroom

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

Director

PizzaBuilder



Pizza

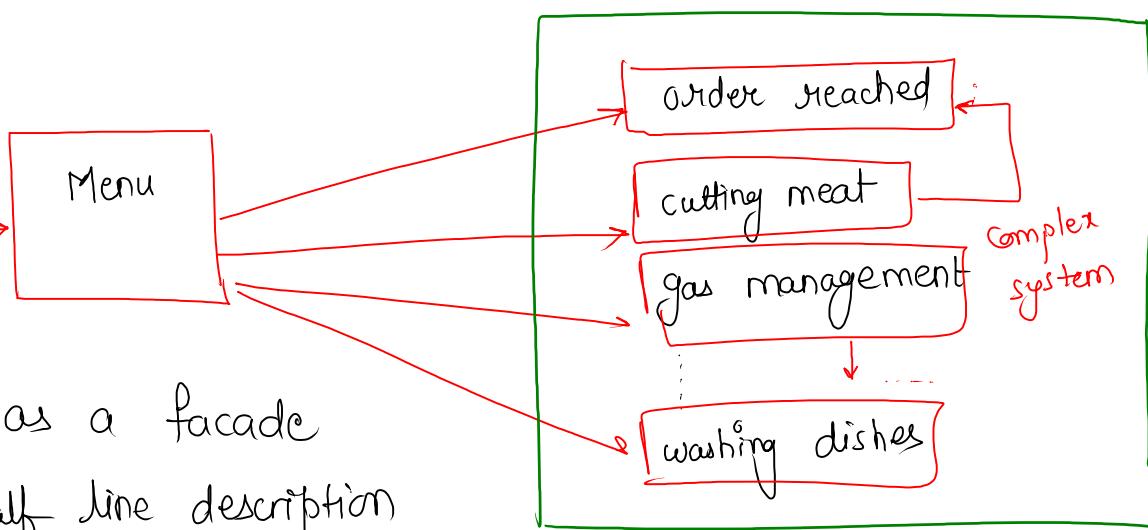
Base base;
Cheese cheese;
Mushroom mushroom;

- - -
- - -
- - -

⇒ Facade design pattern

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

Ex:- like a restraint



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.

Facade scenario 1

```
public class EmployeeClient {  
    public void getEmployeeDetails() {  
        EmployeeFacade employeeFacade = new EmployeeFacade();  
        Employee employeeDetails = employeeFacade.getEmployeeDetails( empID: 121222);  
    }  
}
```



```
public class EmployeeDAO {  
    public void insert(){  
        //insert into Employee Table  
    }  
  
    public void updateEmployeeName(){  
        //updating employee Name  
    }  
  
    public Employee getEmployeeDetails(String emailID){  
        //get employee details based on Emp ID  
        return new Employee();  
    }  
  
    public Employee getEmployeeDetails(int empID){  
        //get employee details based on Emp ID  
        return new Employee();  
    }  
}
```

There could be 50 or 100 methods in EmployeeDAO, but we are exposing only `insert()` to client in EmployeeFacade

```

public class OrderClient {
    public static void main(String args[]){
        OrderFacade orderFacade = new OrderFacade();
        orderFacade.createOrder();
    }
}

```

facade scenario 2

↓ has-a

```

public class OrderFacade {
    ProductDAO productDao;
    Invoice invoice;
    Payment payment;
    SendNotification notification;

    public OrderFacade() {
        productDao = new ProductDAO();
        invoice = new Invoice();
        payment = new Payment();
        notification = new SendNotification();
    }

    public void createOrder() {
        Product product = productDao.getProduct( productId: 121 );
        payment.makePayment();
        invoice.generateInvoice();
        notification.sendNotification();
        //order creation successful
    }
}

```

has-a

```

public class ProductDAO {
    public Product getProduct(int productId){
        //get product based on product id and return
        return new Product();
    }
}

public class Payment {
    public boolean makePayment(){
        //initiate payment and return true if success
        return true;
    }
}

public class Invoice {
    public void generateInvoice(){
        //this will generate the invoice
    }
}

public class SendNotification {
    public void sendNotification(){
        //this will send notification to customer on mobile
    }
}

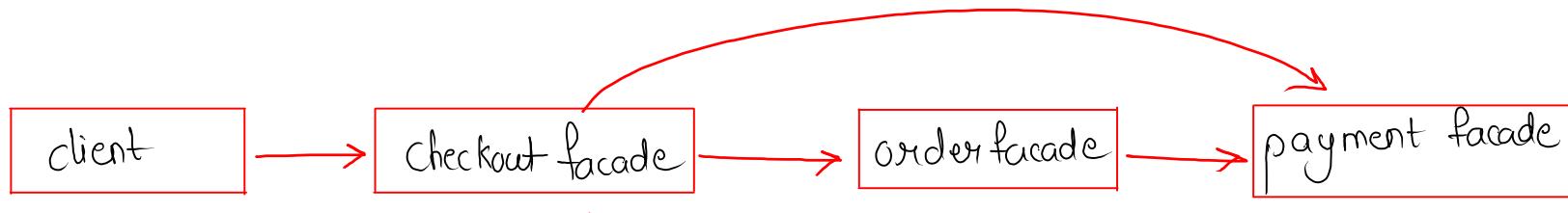
```

here let's suppose we have a system and that system has to be executed in an order (we can't tell client about the order) so we bring a facade layer. and now client don't need to worry , client can only call createOrder()

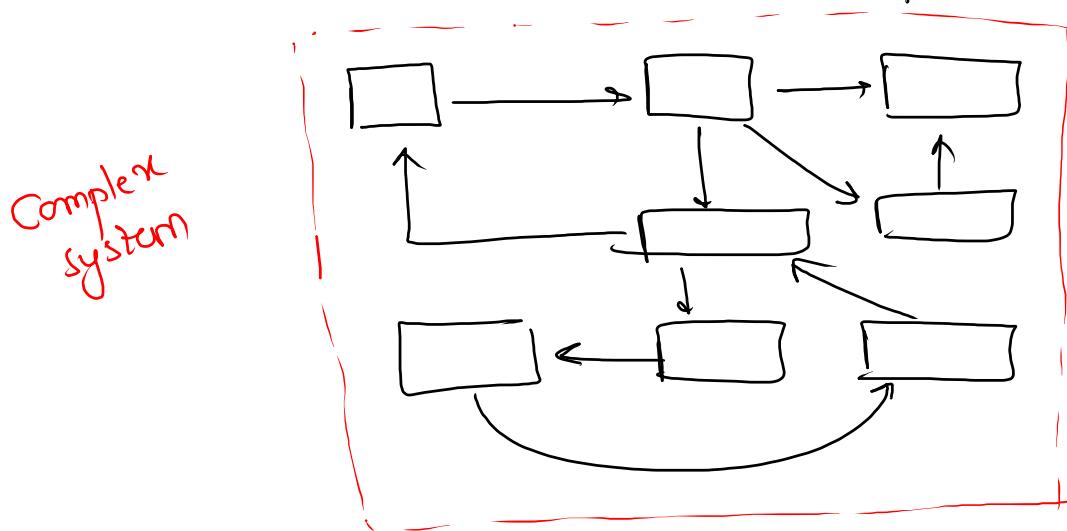
Facade scenario 3

Note: each facade might be having 10-20 or more steps

Facade using other facades



any type of combination is possible



Complex system

⇒ Facade vs Proxy

bcz client calling → facade calling → real obj

& client calling → proxy calling → real obj

Ans:- proxy can work a one particular obj and proxy implements the same interface which is implemented by real obj.

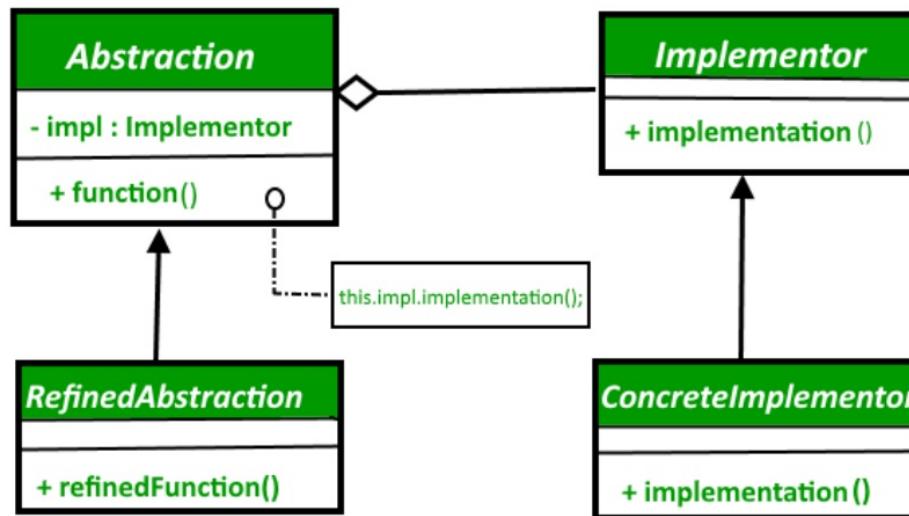
⇒ Facade vs Adapter

In adapter, client and real obj was not compatible with each other which is not the case with facade.

⇒ Bridge design pattern

It decouples the abstraction with its implementation so that the two can vary independently.

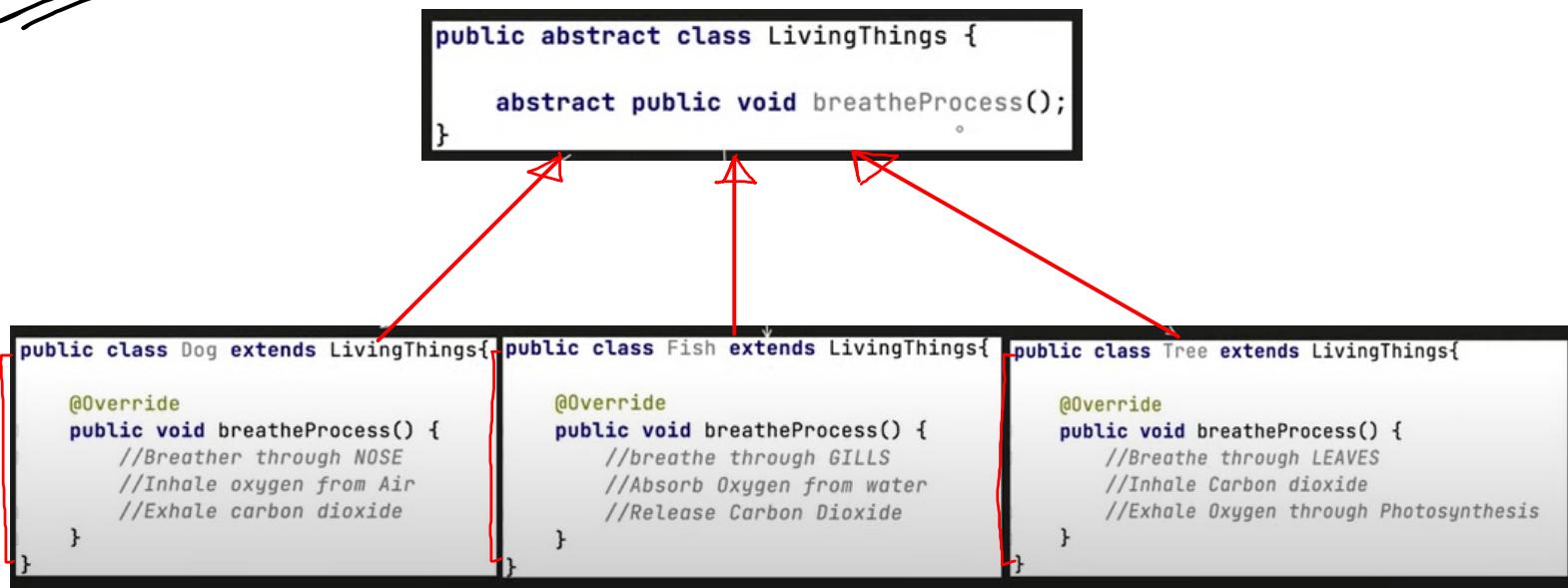
UML



Note:- bridge and strategy design pattern are very similar, even the UML is same

Only difference is in intention while implementation. Intention in bridge is that both Implementation and Abstractor can grow independently.

Problem

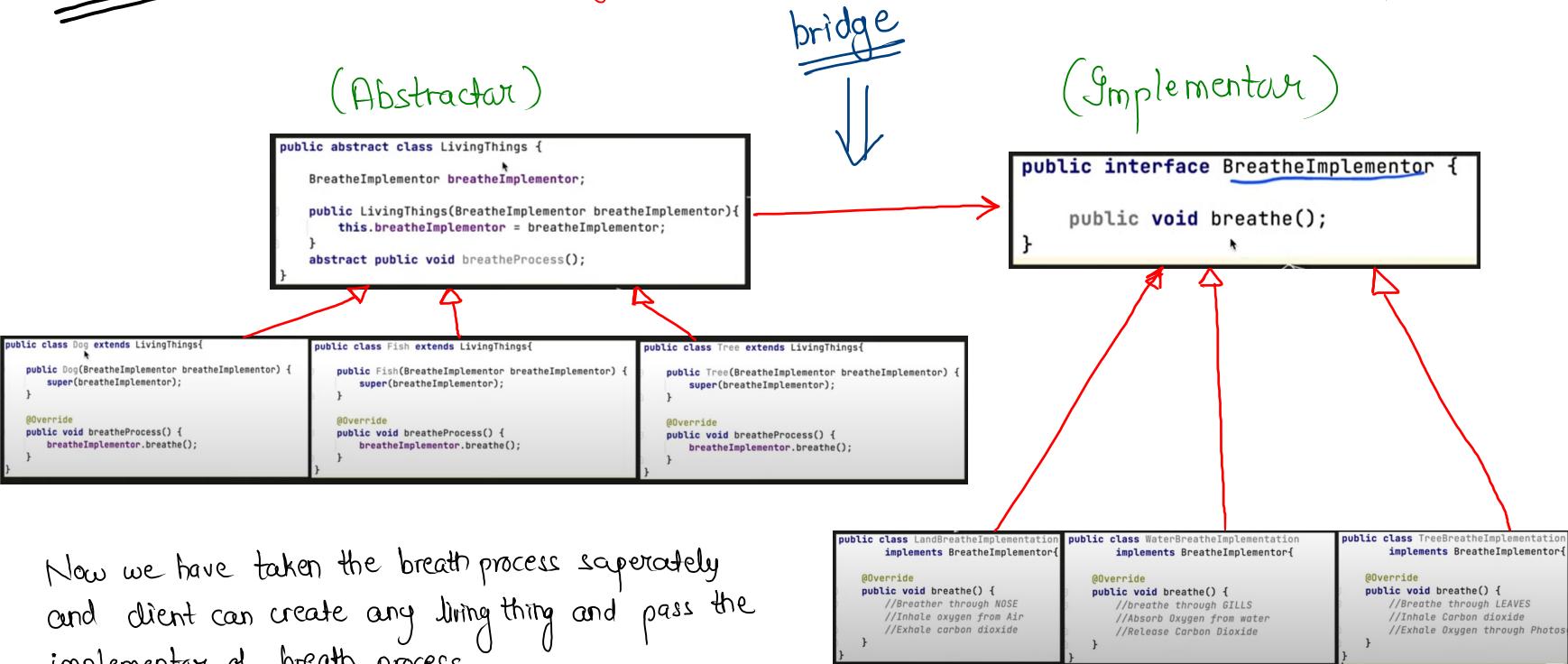


Now, here the problem is if I want to implement another class for Birds , then I have to create another class and then extend it.

But why should I implement whole `breatheProcess()` again , why can't I use the same function again.

Solution

Now we can keep adding more classes in both abstractor and implementor separately.



Now we have taken the breath process separately
and client can create any living thing and pass the
implementor of breath process

now we can use LandBreathImpl any time for cat,
dog, cow, sheep etc. etc. and we don't have to implement
it again and again.

(these are the objects which we are ultimately
using)

Client code

```
LivingThings fishObject = new Fish(new WaterBreathImplementation());
fishObject.breatheProcess();
```

⇒ Prototype Design Pattern

This pattern is used when we want to copy/clone an object, because creating the original object again with minor changes is very expensive.

Problem

```
public class Student {  
    int age;  
    private int rollNumber;    some fields are  
    String name;            private  
  
    Student(){  
    }  
  
    Student(int age, int rollNumber, String name){  
        this.age = age;  
        this.rollNumber = rollNumber;  
        this.name = name;  
    }  
}
```

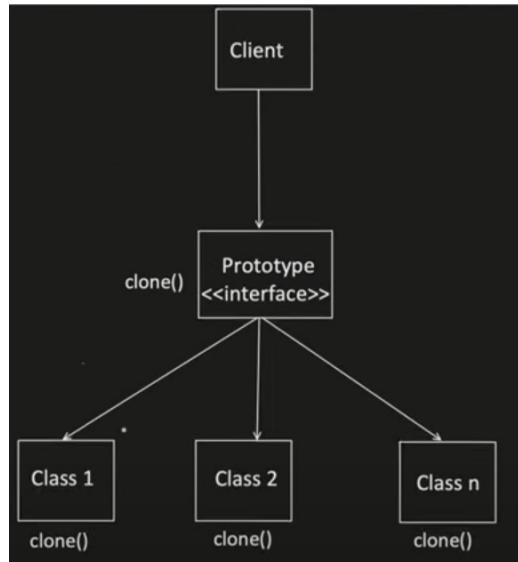
```
public class Main {  
  
    public static void main(String args[]){  
        Student obj = new Student( age: 20, rollNumber: 76, name: "Ram");  
        //original obj  
        Student cloneObj = new Student();  
        cloneObj.name = obj.name;  
        cloneObj.age = obj.age;  
        cloneObj.rollNumber = obj.rollNumber;  
    }  
}
```

problem

here we can't copy roll no. bcz it is private
and also to copy every field, we need to know
about the original class entirely.

That is why cloning should not be the task for client
and class should have do it itself.

UML



Note:- client is only calling .clone
method directly
and clone method is implemented
in original class itself
where we can even access
the private fields.

```
public class Main {  
    public static void main(String args[]){  
        Student obj = new Student( age: 20, rollNumber: 75, name: "Ram");  
        Student cloneObj = (Student) obj.clone();  
    }  
}
```

```
public interface Prototype {  
    Prototype clone();  
}
```

```
public class Student implements Prototype{  
    int age;  
    private int rollNumber;  
    String name;  
  
    Student(){  
    }  
  
    Student(int age, int rollNumber, String name){  
        this.age = age;  
        this.rollNumber = rollNumber;  
        this.name = name;  
    }  
  
    @Override  
    public Prototype clone() {  
        return new Student(age, rollNumber, name);  
    }  
}
```

The code shows the implementation of the Prototype pattern. It includes a 'Main' class with a 'main' method that creates a 'Student' object and clones it. It also includes the 'Prototype' interface with a single 'clone()' method. Finally, it shows the 'Student' class which implements 'Prototype' and overrides the 'clone()' method to return a new instance of itself with the same values.

⇒ Singleton design pattern

It is used when we have to create only 1 instance of the class.

Ways to achieve this

- 1) Eager
- 2) Lazy
- 3) Synchronized
- 4) Double Locking



→ Eager Initialization (first of all constructor should be private, so that no one outside the class should call constructor to initialize)

```
public class DBConnection {  
    private static DBConnection conObject = new DBConnection();  
  
    private DBConnection(){  
    }  
  
    public static DBConnection getInstance(){  
        return conObject;  
    }  
}
```

here, whenever we create an object in client code then every time an object is being created.

(even when it is not reqd)

which is simply memory

wastage

and object

```
public class Main {  
    public static void main(String args[]){  
        DBConnection connObject = DBConnection.getInstance();  
    }  
}
```

Note:- we have created getInstance() as static so that other classes can call it. bcz every static thing belongs to class level.

→ Lazy Initialization

```
public class DBConnection {  
    private static DBConnection conObject;  
    private DBConnection(){  
    }  
  
    public static DBConnection getInstance(){  
        if(conObject == null){  
            conObject = new DBConnection();  
        }  
        return conObject;  
    }  
}
```

↑
null initially

In lazy initialization, we are not creating an object everytime, we are only creating it for the first time and then only returning same obj.

But, what if 2 thread come at the exact same time then it will create object two times bcz both threads found obj as null.
we have resolved it using synchronised

→ Synchronised method

```
public class DBConnection {  
    private static DBConnection conObject;  
    private DBConnection(){  
    }  
  
    synchronized public static DBConnection getInstance(){  
        if(conObject == null){  
            conObject = new DBConnection();  
        }  
        return conObject;  
    }  
}
```

now, if two threads come at same time then synchronised will send 1 and lock the other one and then later send the next one.

But, what if 1000 threads are coming for getInstance then 1st one will be send and 999 will be locked one by one.

Locking is a very expensive and time consuming task, that is why this is not feasible.

→ Double docking (used in industry)

```
public class DBConnection {  
    private static DBConnection conObject;  
  
    private DBConnection(){  
    }  
  
    public static DBConnection getInstance(){  
        if(conObject == null){  
            synchronized (DBConnection.class){  
                if(conObject == null){  
                    conObject = new DBConnection();  
                }  
            }  
        }  
        return conObject;  
    }  
}
```

Now, if 1000 threads are coming at the same time then only first two will enter first check but synchronised will let first one go inside which will initialise the object until then 2nd one is locked, later on 2nd one will be released and found object already declared so it will not enter the inner check.

Now, all other 998 threads will not be able to enter even the first check which saved us locking process for all these 998 threads.