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 responsibity, Command, Interpreter, mediator, observer, state, strategy, template, visitor)

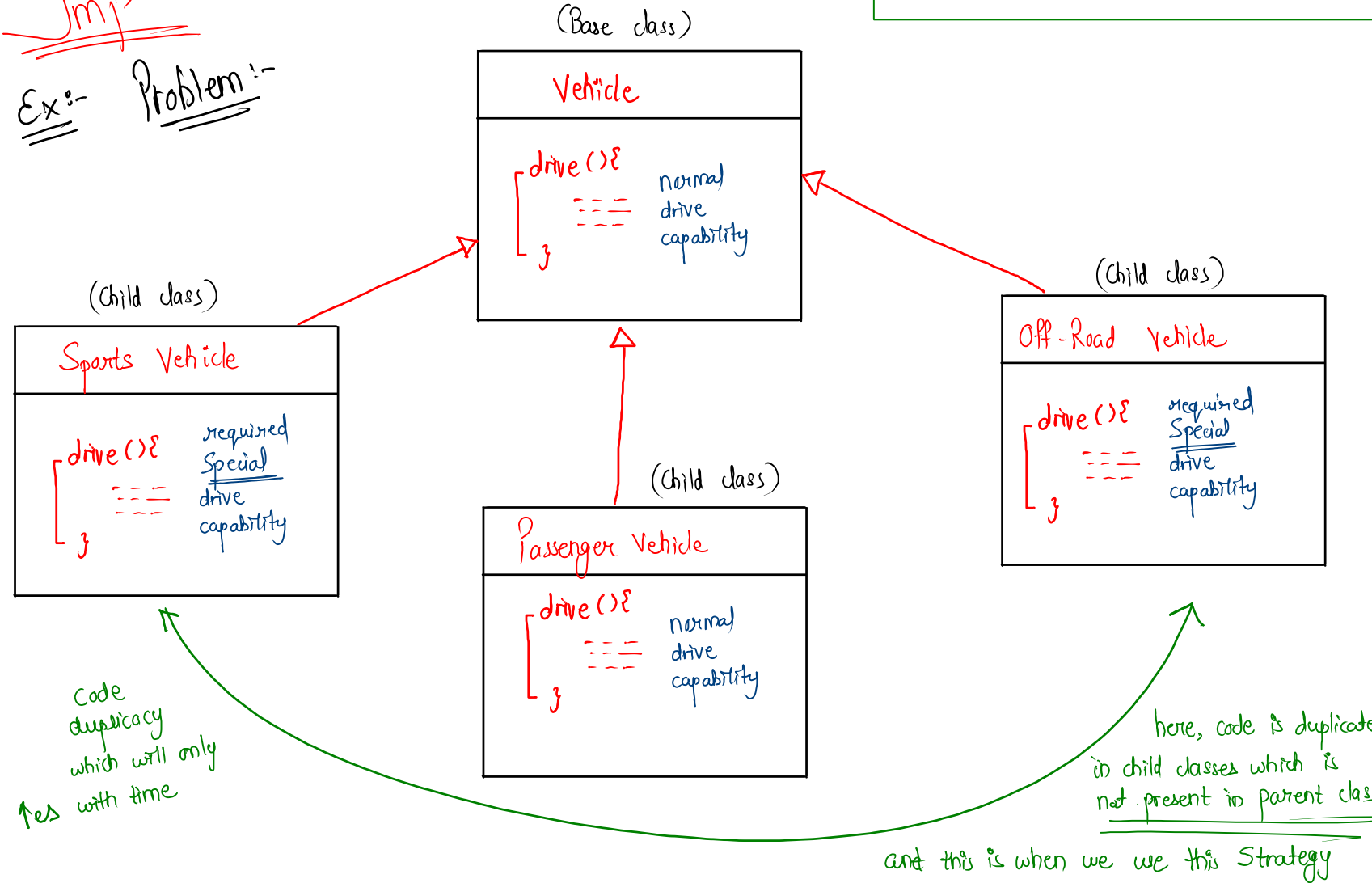
⇒ Strategy Design Pattern

Imp

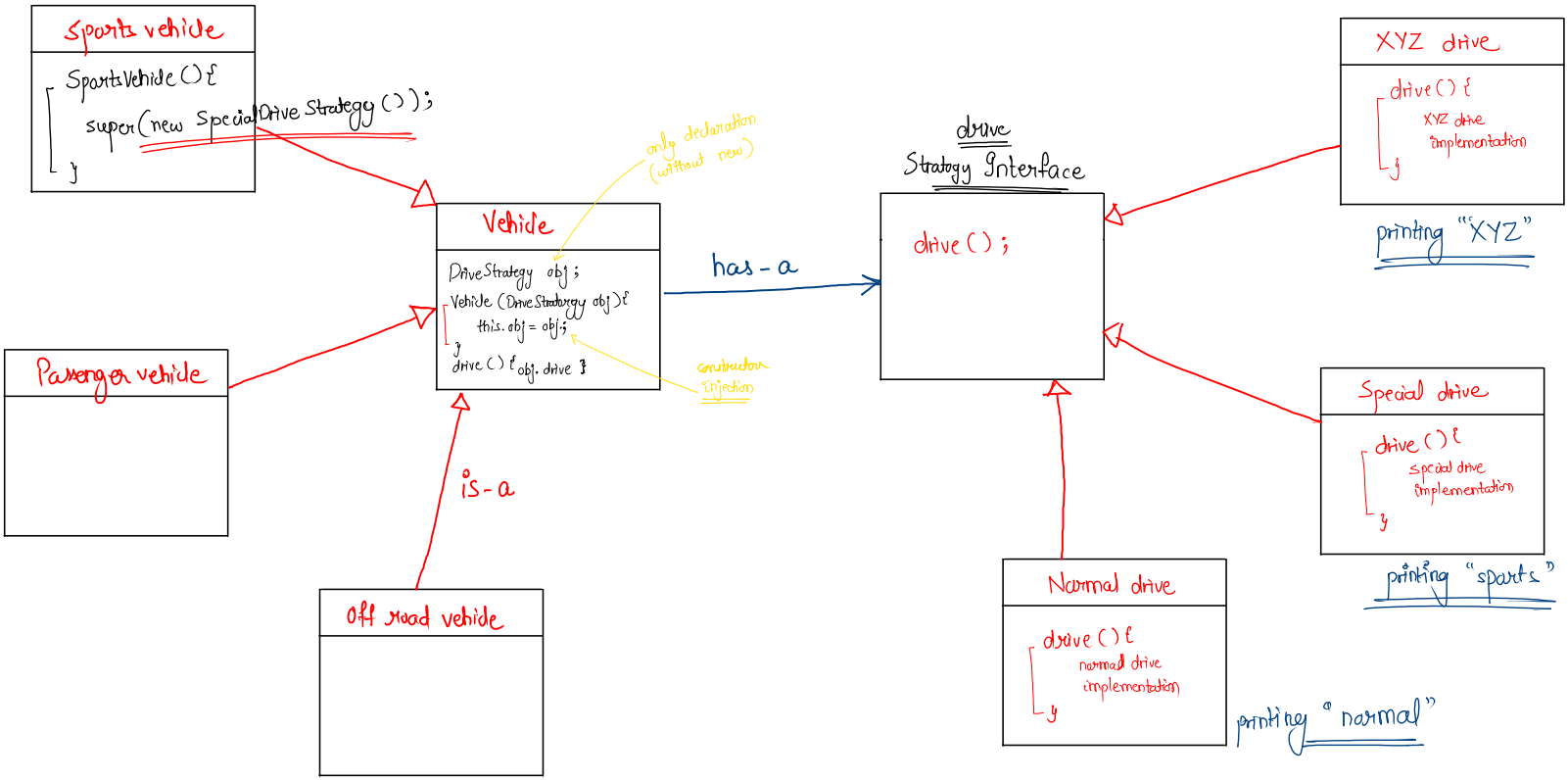
Ex:- Problem :-

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

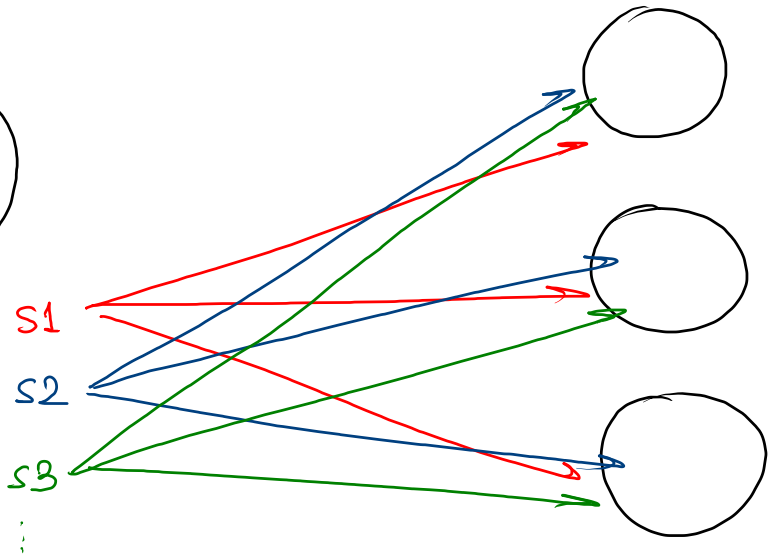
In amazon.com, we are looking for a product which is unavailable and there is a button of notify me!, so send notifications to customers when product is available.

Implement this button (LLD questions)

Note:- There are 2 states

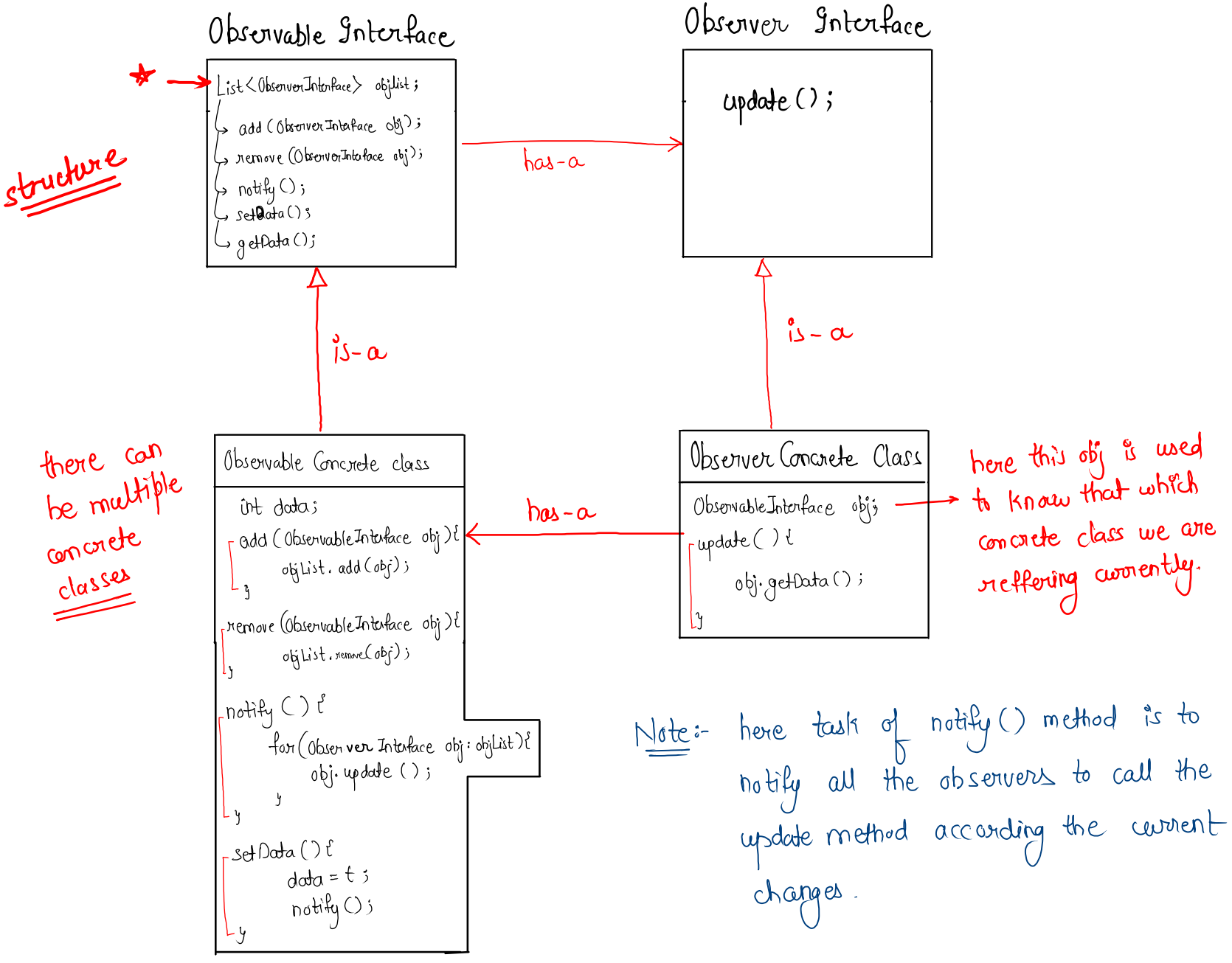
observable

state change



multiple observers

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

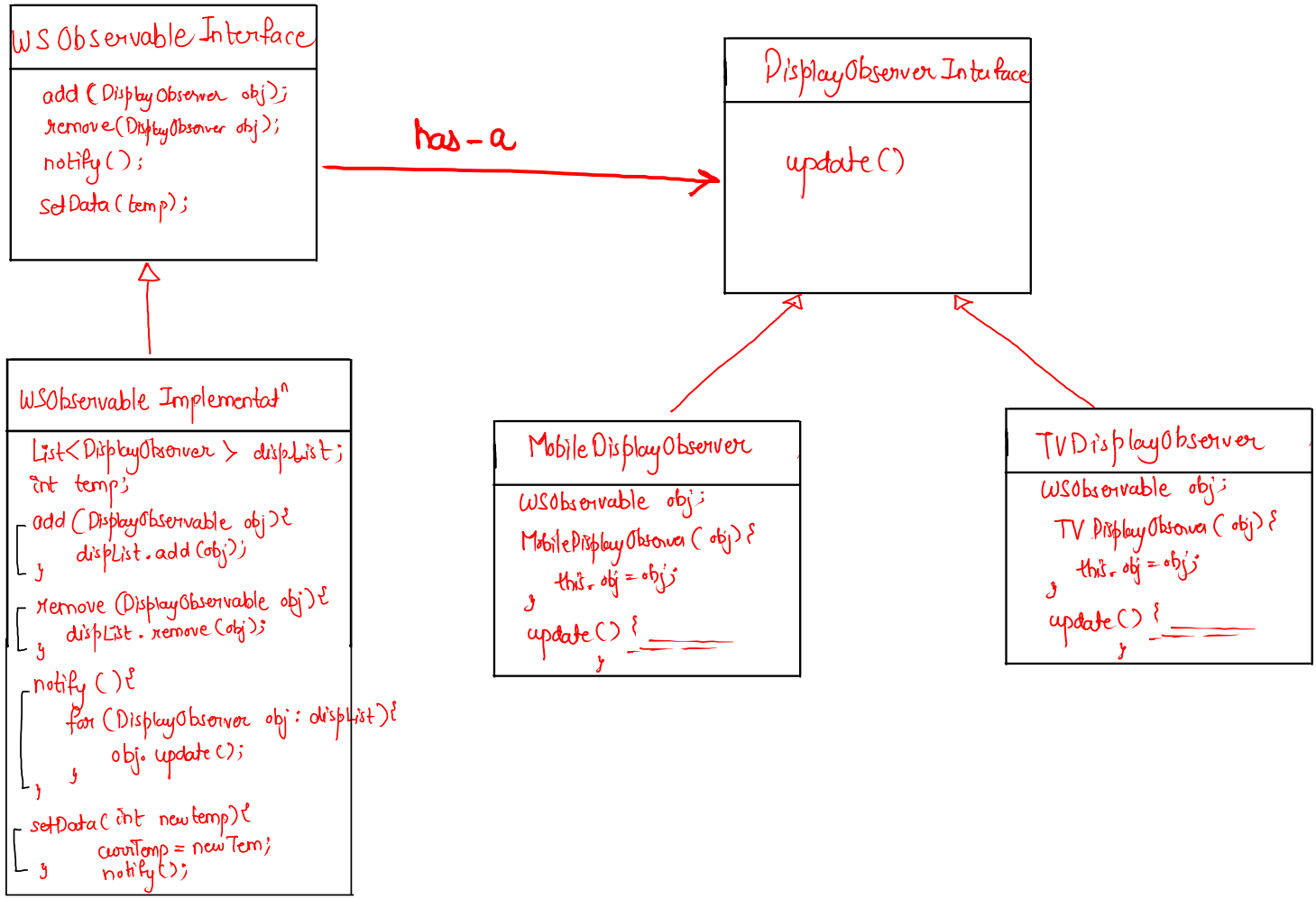

Observable Interface
List<ObserverInterface> objList;
add(ObserverInterface obj);
remove(ObserverInterface obj);
notify();
setData();
getData();
Observer Interface
update();
has-a
is-a
is-a
structure
there can be multiple concrete classes
Observable Concrete class
int data;
add(ObservableInterface obj){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

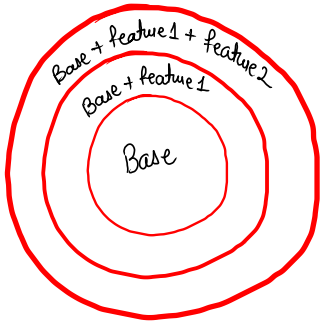
client code

```
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)

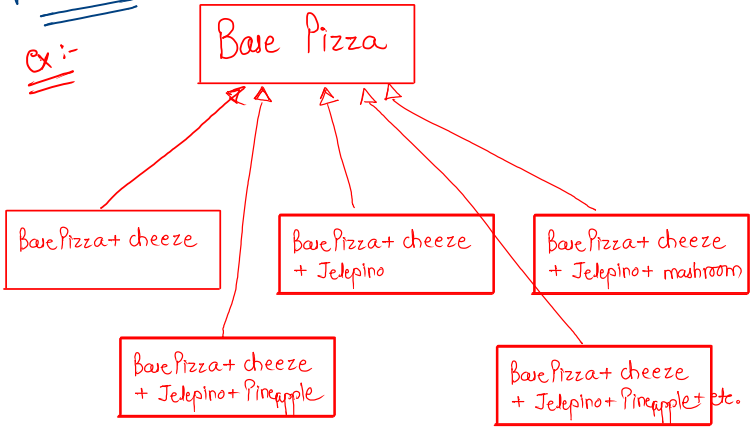


Why do we need decaratar pattern?

↳ 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;
    }
}
```

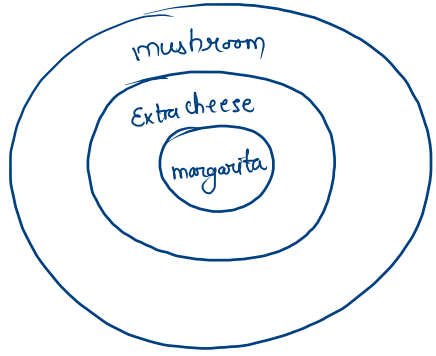
```java
public class Margherita extends BasePizza{

    @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

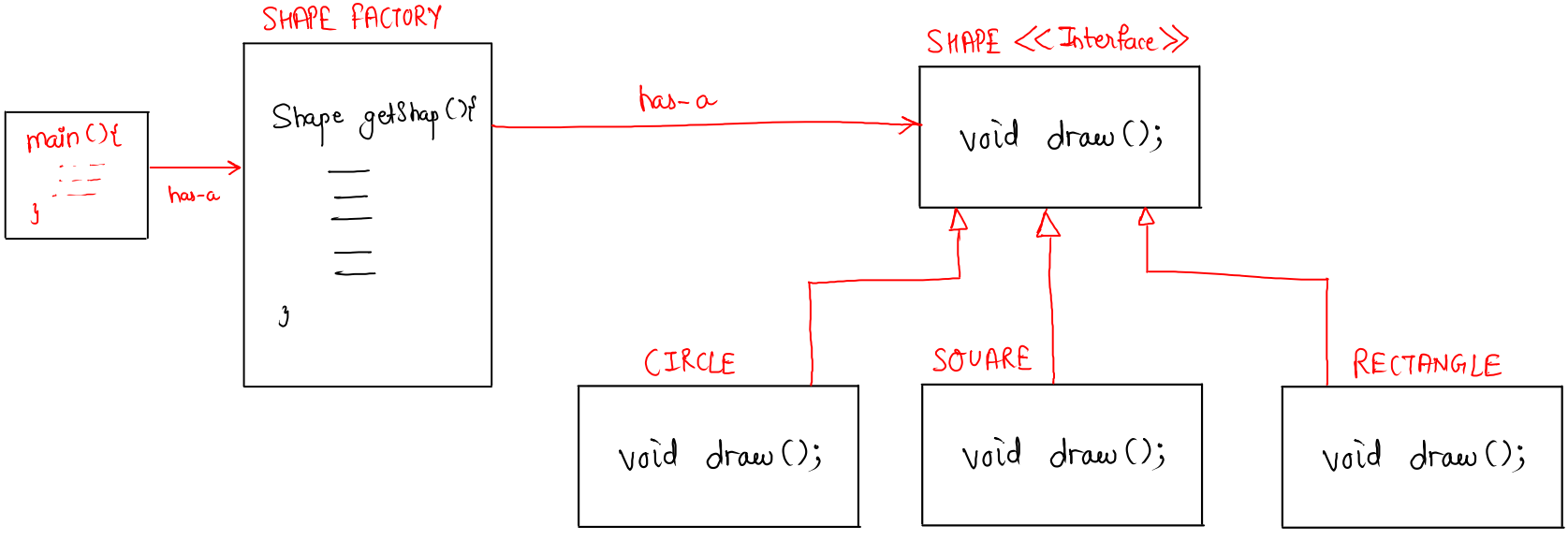
```java
Pizza pizza = new ExtraCheese(new Margarita());
pizza.cost()  // 100 + 10 = 110
```




```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();
}
```

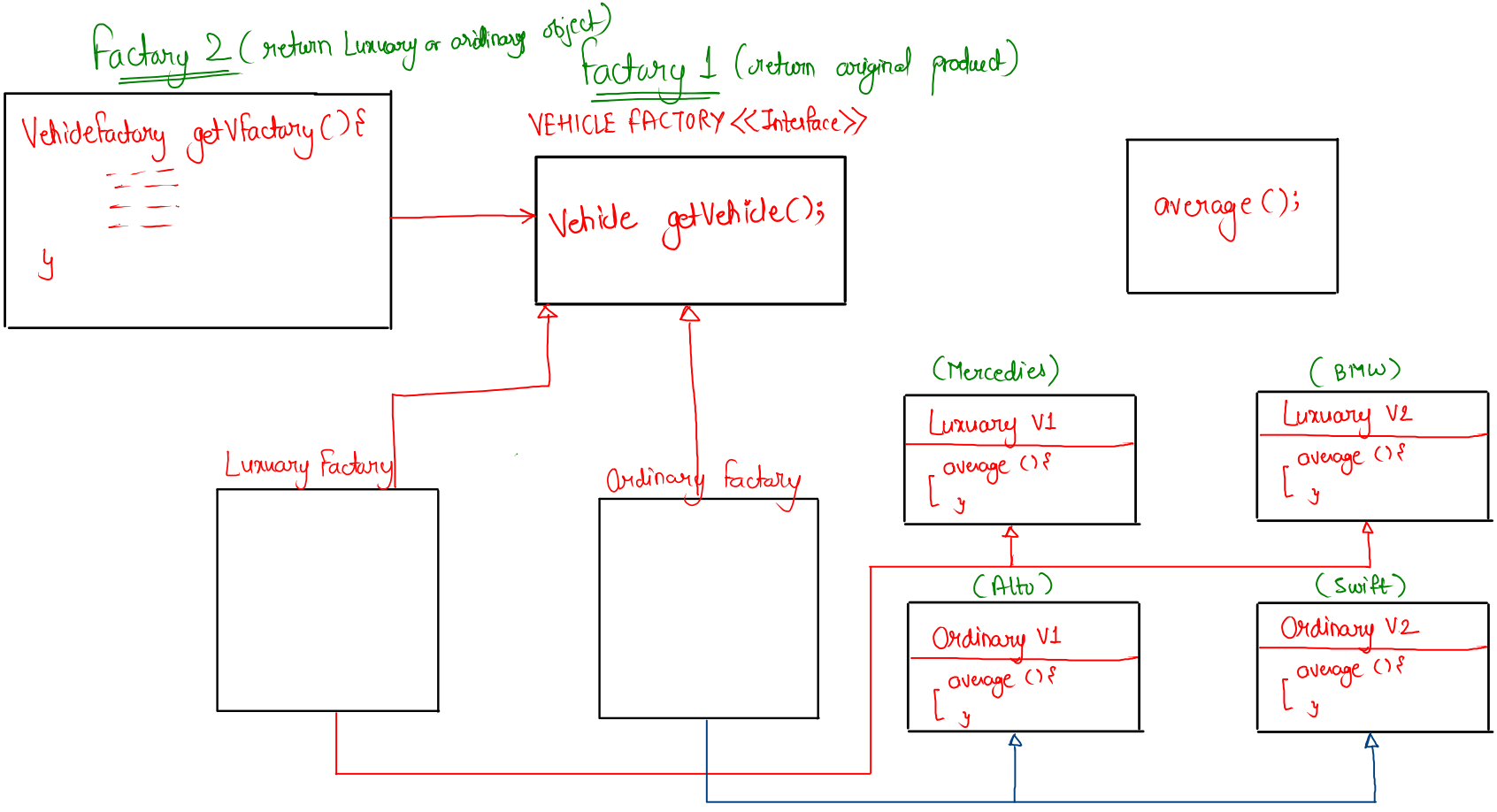
```java
public class Rectangle implements Shape{

    @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

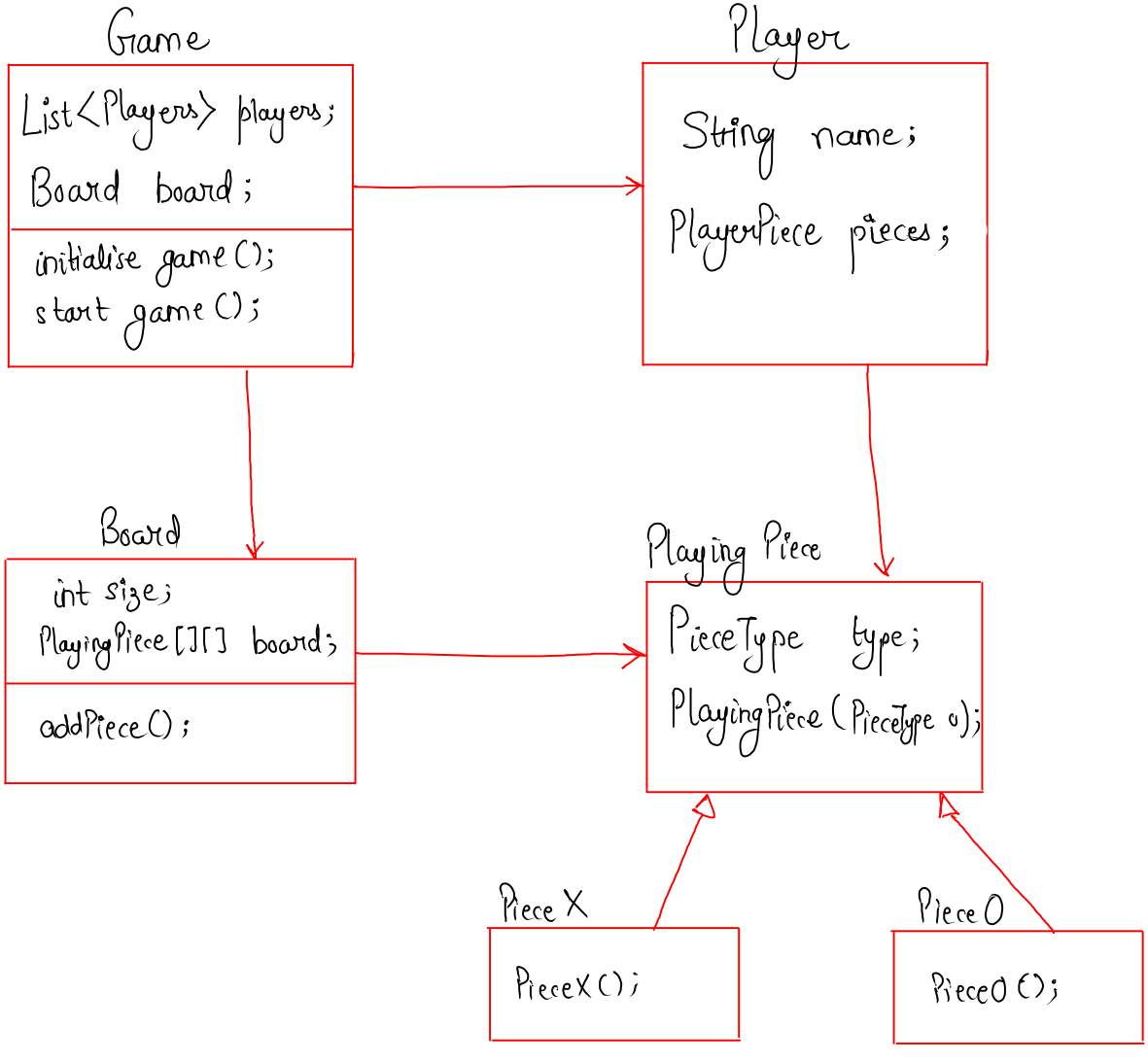
⇒ Abstract Factory Pattern :- (It's a Factory of factory)

↳ 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 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 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 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());
    }
}
```

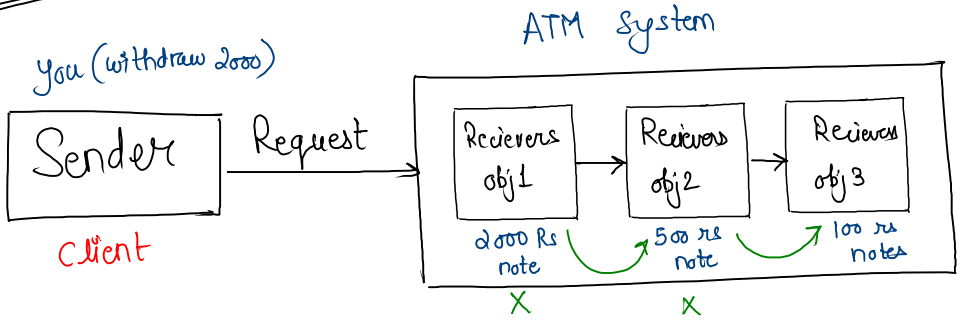
⇒ Chain of responsibility Design Pattern

Application usage :-

- ATM / Vending machine
- Design logger (Amazon)

structure

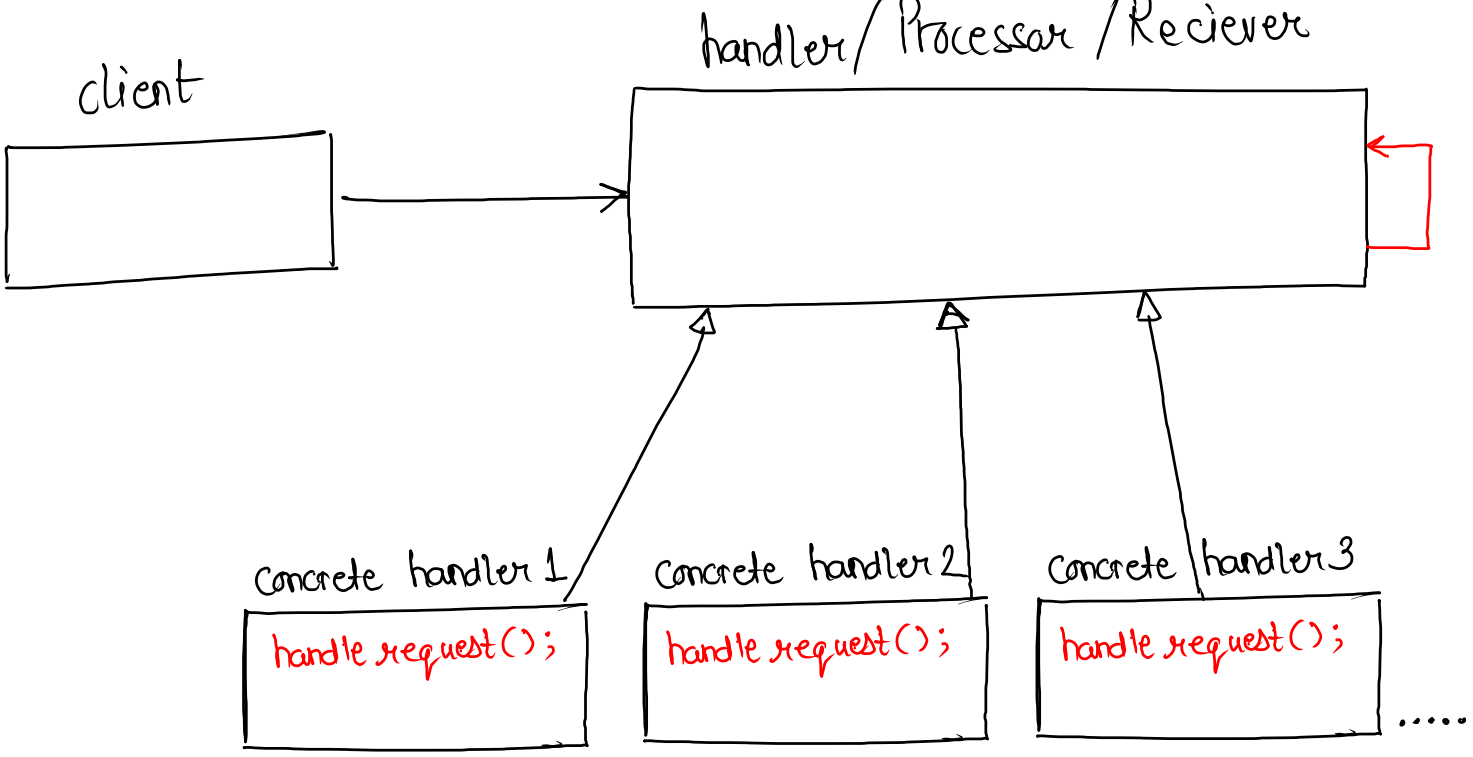
Ex:- You (withdraw 2000)



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);
        }

    }
}
```

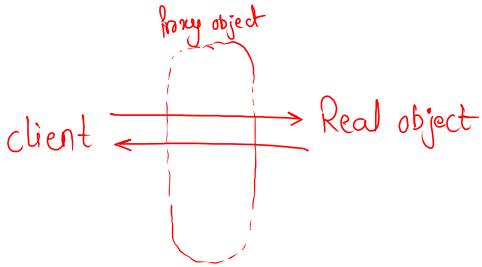
```java
public class ErrorLogProcessor extends LogProcessor{

    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

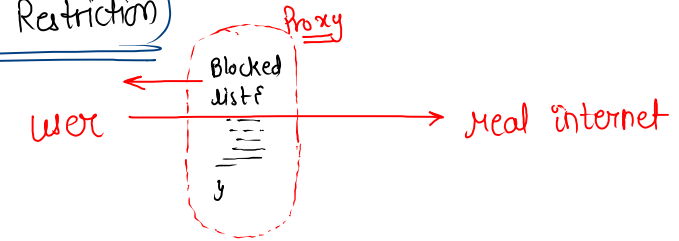
Ex1



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

(Internet Restriction)

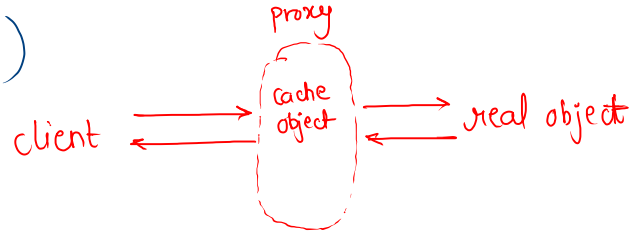
Ex2



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

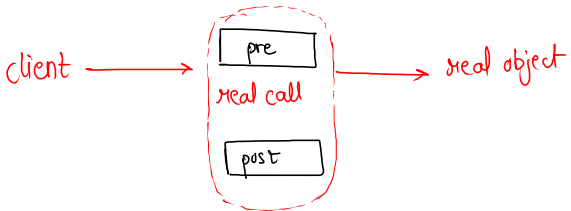
(Caching)

Ex3



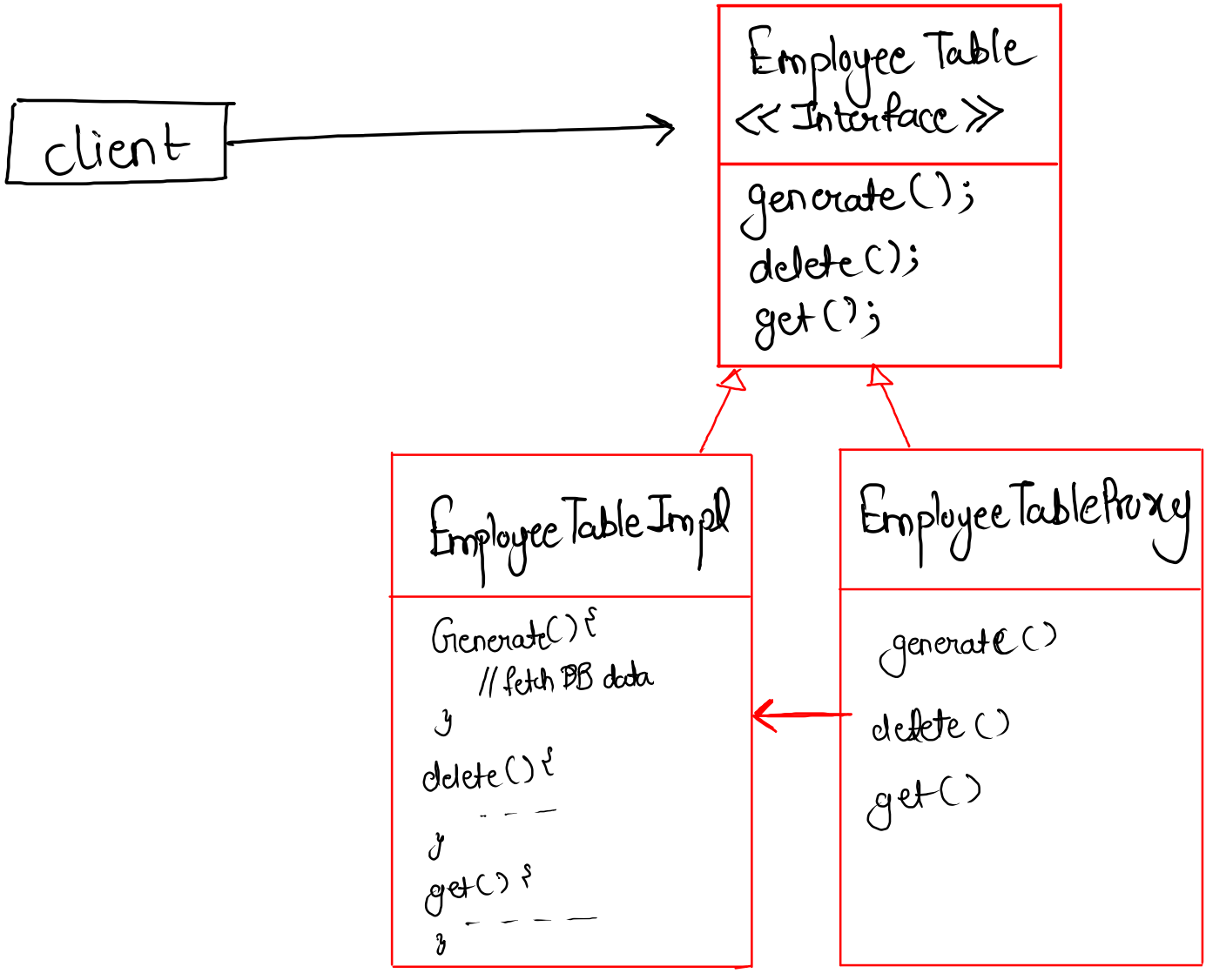
Caching is another example of proxy design pattern, because here as well we first have a proxy in b/w client & real object to check does cache already have the result.

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;
}
```

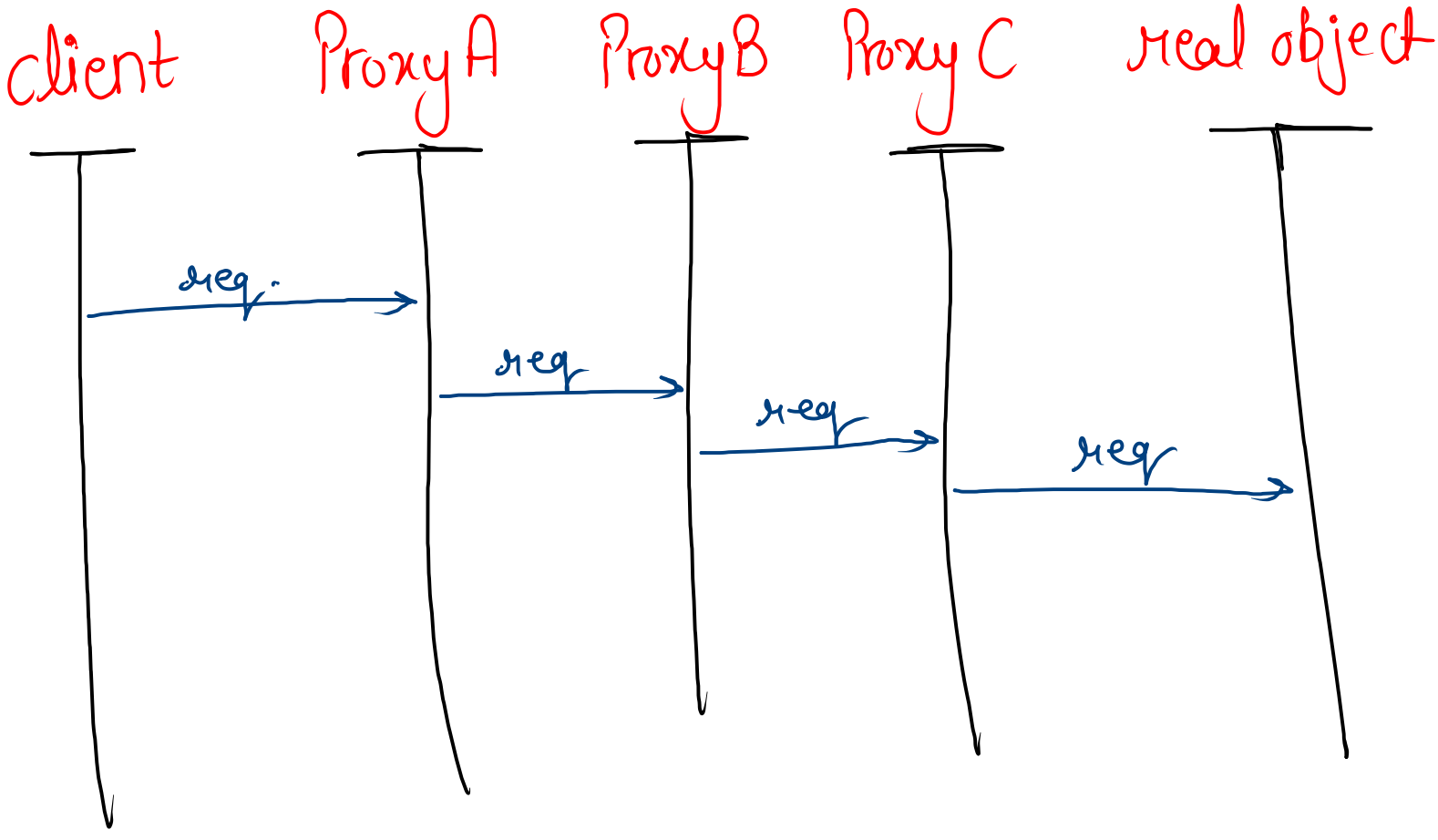
```java
public class EmployeeDaoImpl implements EmployeeDao{

    @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());
}
```

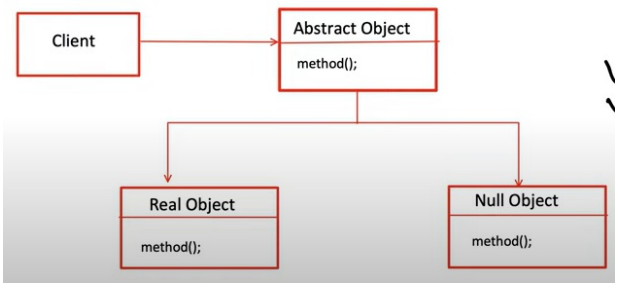
Solution :- But we can't put this check in every single place.

```
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

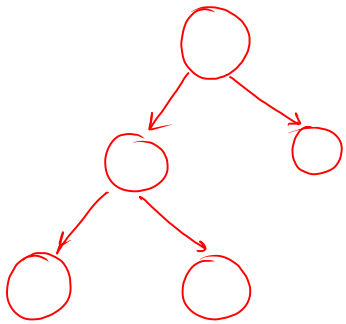
Note:- now we are returning a null object which are reflecting a default behaviour instead of giving error back.

```
public class Car implements Vehicle{

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

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

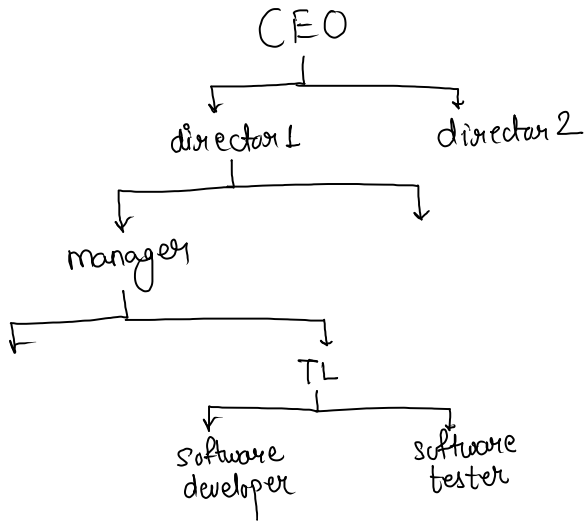
```
public class NullVehicle implements Vehicle{

    @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

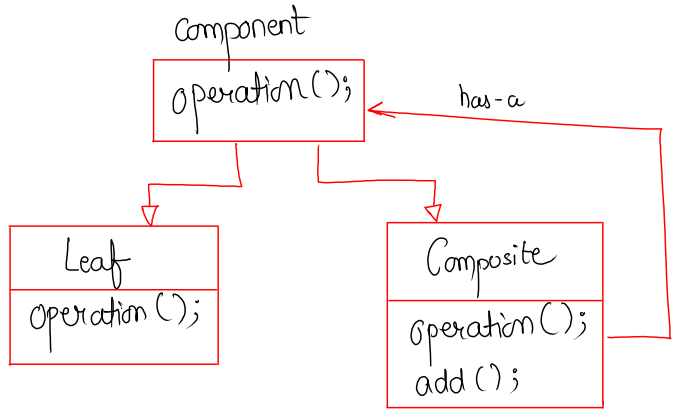
↓

file or directory

↓

file or directory

UML diagram



leaf node of tree

Composite object is which is containg object of itself

client code

```
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);
    }
}
```

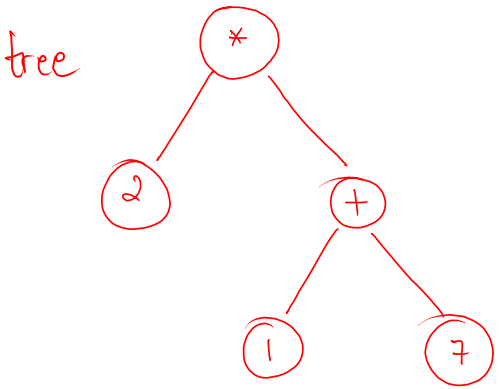
```
public class Directory implements FileSystem {
    String directoryName;
    List<FileSystem> fileSystemList;

    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

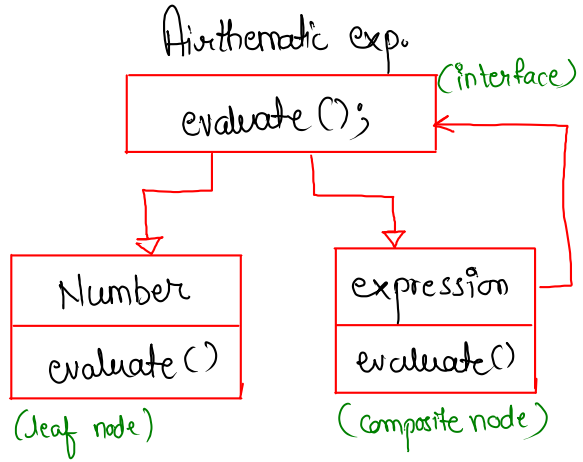
now we don't need to check if-else cond^n here, because ls() funtion is present in the interface itself

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;
    }
}
```

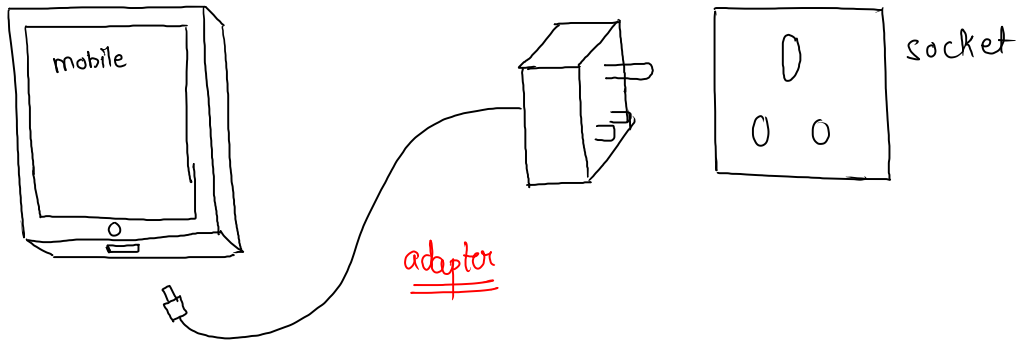
```java
ArithmeticExpression two = new Number( value: 2);

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

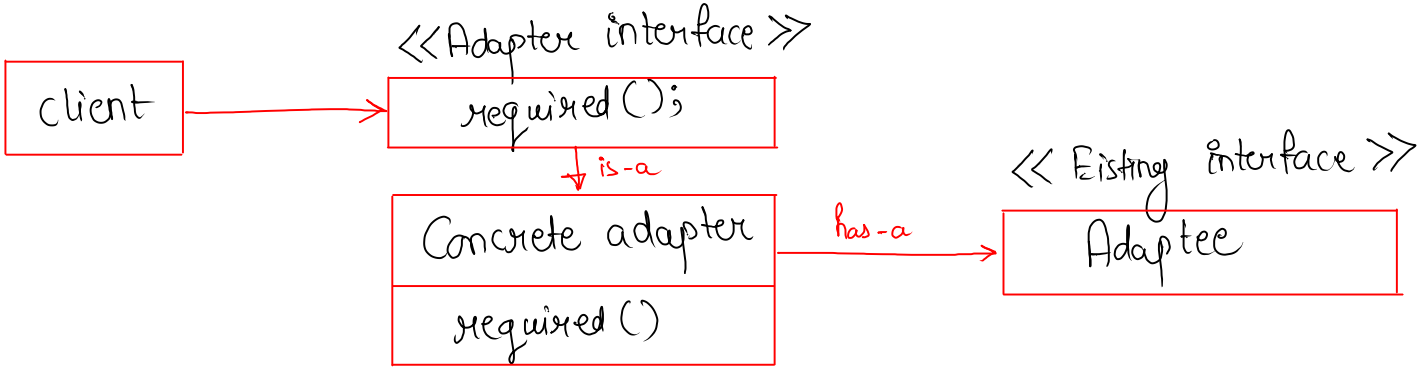
It is a bridge b/w existing interface & expected interface

Ex1

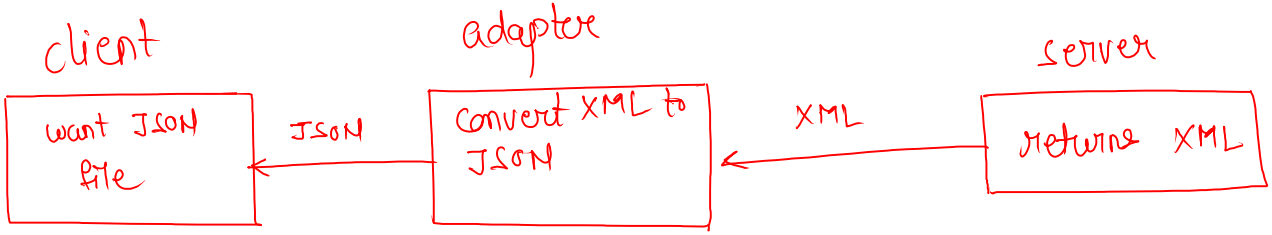


(here mobile is not compatible to directly get charging from socket, so we need a adapter)

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

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

adapter

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

client wanted in kg

```java
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

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

```java
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.

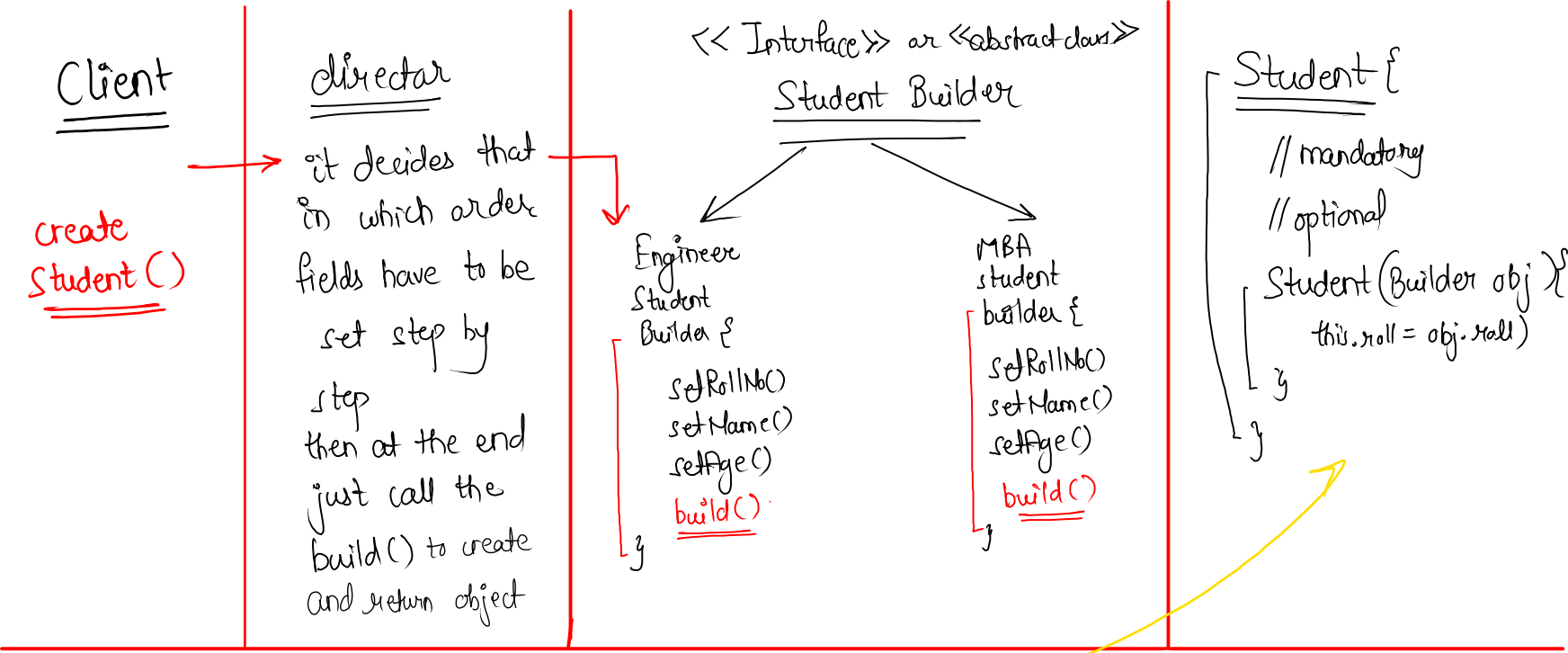
Problem

```
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());

    }
}
```

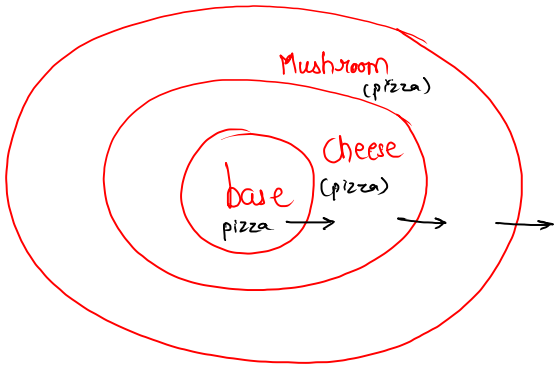
client code become easy

```
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

Base Pizza + Cheeze Builder {
- addCrust()
- addCheese()
- ---
- build()

}

Base Pizza + Mushroom Builder {
- addCrust()
- addMushroom()
- ---
- build()

}

Pizza {

Base base;

Cheese cheese;

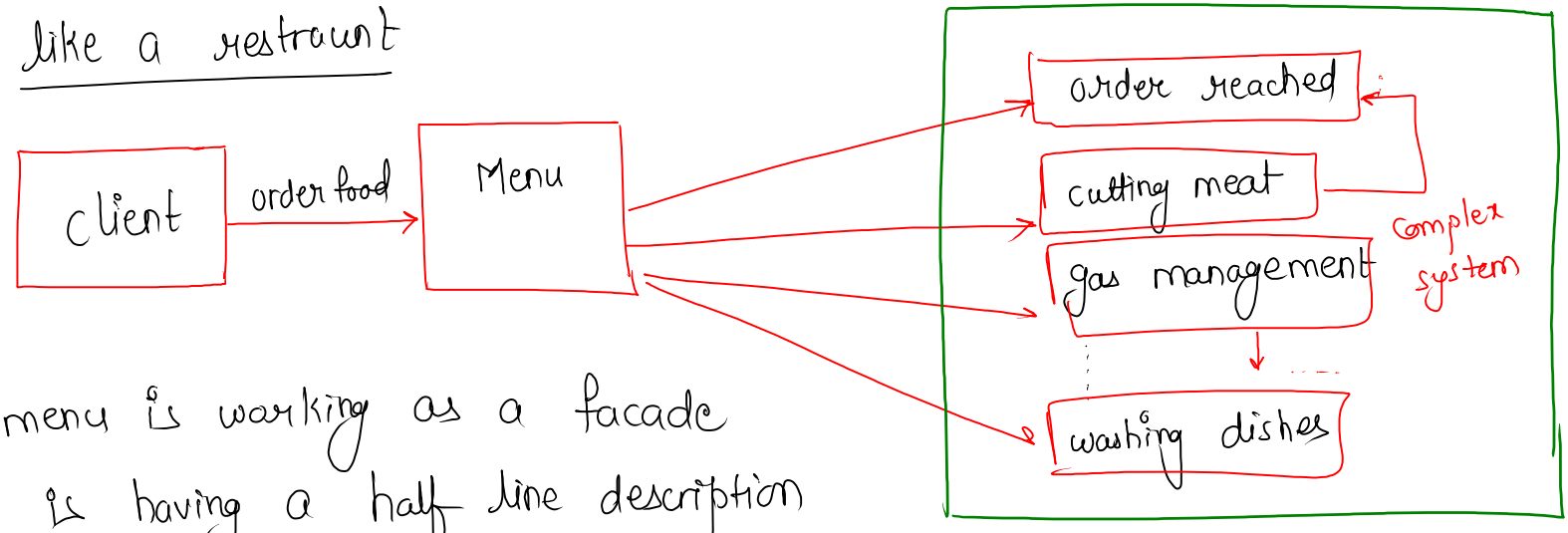
Mushroom mushroom;

}

⇒ 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.

facade scenario 1

```java
public class EmployeeClient {

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

```java
public class EmployeeFacade {

    EmployeeDAO employeeDAO;

    public EmployeeFacade(){
        employeeDAO = new EmployeeDAO();
    }

    public void insert(){
        employeeDAO.insert();
    }

    public Employee getEmployeeDetails(int empID){
        return employeeDAO.getEmployeeDetails(empID);
    }
}
```

```java
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

facade scenario 2

```java
public class OrderClient {

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

has-a

```java
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 successfull
    }
}
```

has-a

```java
public class ProductDAO {

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

```java
public class Payment {

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

```java
public class Invoice {

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

```java
public class SendNotification {

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

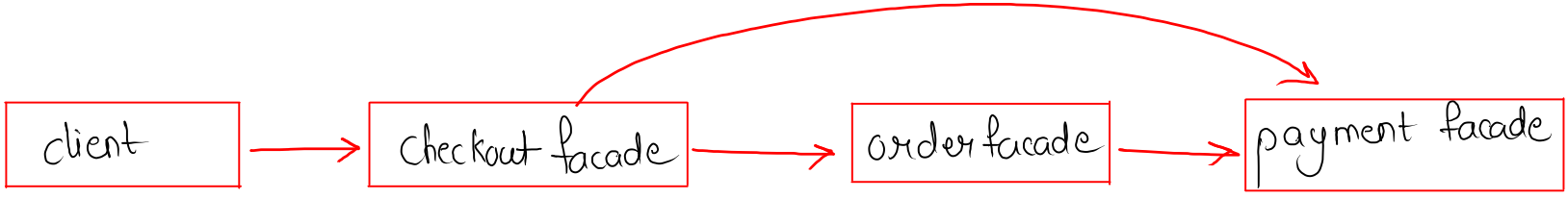
here let's suppose we have a system and that system has to be executed in an order (we can't tell client about the order) so we bring a facade layer.

and now client don't need to worry, client can only call createOrder()

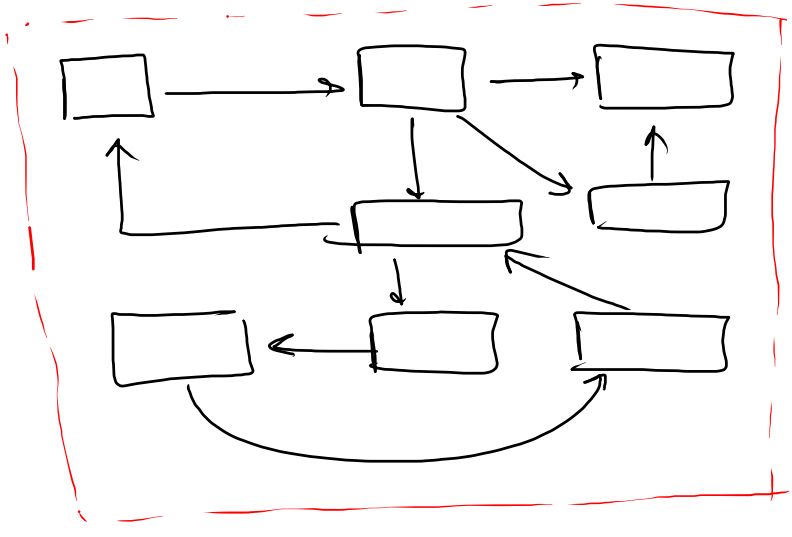
facade scenario 3

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

facade using other facades



any type of combination is possible



⇒ Facade vs Proxy

bcz client —calling→ facade —calling→ real obj

& client —calling→ Proxy —calling→ real obj

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

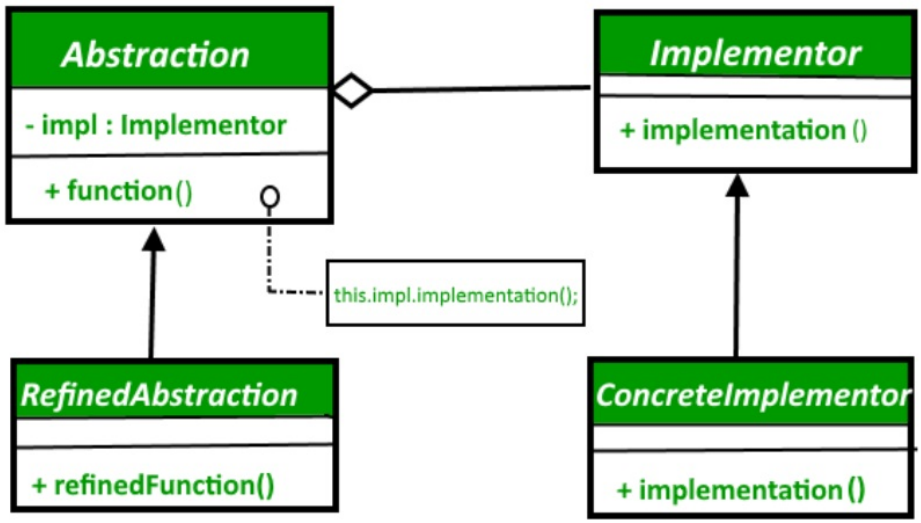
⇒ Facade vs Adapter

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.

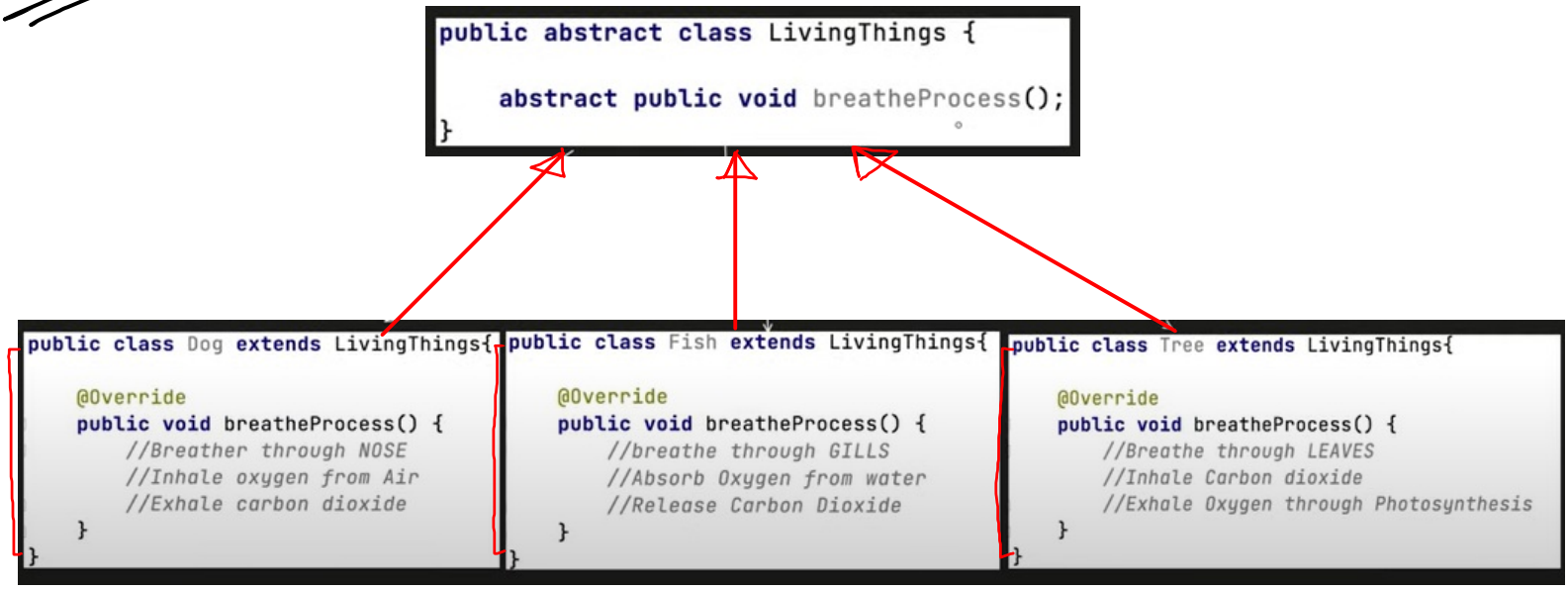
UML



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

Only difference is in Intention while implemention. Intention in bridge is that both Implementar and Abstractar can grow independently.

Problem



```java
public abstract class LivingThings {

    abstract public void breatheProcess();
}
```

```java
public class Dog extends LivingThings{

    @Override
    public void breatheProcess() {
        //Breather through NOSE
        //Inhale oxygen from Air
        //Exhale carbon dioxide
    }
}
```

```java
public class Fish extends LivingThings{

    @Override
    public void breatheProcess() {
        //breathe through GILLS
        //Absorb Oxygen from water
        //Release Carbon Dioxide
    }
}
```

```java
public class Tree extends LivingThings{

    @Override
    public void breatheProcess() {
        //Breathe through LEAVES
        //Inhale Carbon dioxide
        //Exhale Oxygen through Photosynthesis
    }
}
```

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 breathProcess() again, why can't I use the same function again.

Solution Now we can keep adding more classes in both abstractar and implementar seperately.

bridge

(Abstractar)

```
public abstract class LivingThings {

    BreatheImplementor breatheImplementor;

    public LivingThings(BreatheImplementor breatheImplementor){
        this.breatheImplementor = breatheImplementor;
    }
    abstract public void breatheProcess();
}
```

(Implementar)

```
public interface BreatheImplementor {

    public void breathe();
}
```

```
public class Dog extends LivingThings{

    public Dog(BreatheImplementor breatheImplementor) {
        super(breatheImplementor);
    }

    @Override
    public void breatheProcess() {
        breatheImplementor.breathe();
    }
}
```

```
public class Fish extends LivingThings{

    public Fish(BreatheImplementor breatheImplementor) {
        super(breatheImplementor);
    }

    @Override
    public void breatheProcess() {
        breatheImplementor.breathe();
    }
}
```

```
public class Tree extends LivingThings{

    public Tree(BreatheImplementor breatheImplementor) {
        super(breatheImplementor);
    }

    @Override
    public void breatheProcess() {
        breatheImplementor.breathe();
    }
}
```

```
public class LandBreatheImplementation
        implements BreatheImplementor{

    @Override
    public void breathe() {
        //Breather through NOSE
        //Inhale oxygen from Air
        //Exhale carbon dioxide
    }
}
```

```
public class WaterBreatheImplementation
        implements BreatheImplementor{

    @Override
    public void breathe() {
        //breathe through GILLS
        //Absorb Oxygen from water
        //Release Carbon Dioxide
    }
}
```

```
public class TreeBreatheImplementation
        implements BreatheImplementor{

    @Override
    public void breathe() {
        //Breathe through LEAVES
        //Inhale Carbon dioxide
        //Exhale Oxygen through Photos
    }
}
```

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

now we can we 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 WaterBreatheImplementation());
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

```java
public class Student {

    int age;
    private int rollNumber;
    String name;

    Student(){
    }

    Student(int age, int rollNumber, String name){
        this.age = age;
        this.rollNumber = rollNumber;
        this.name = name;
    }
}
```

some fields are private

```java
public class Main {

    public static void main(String args[]) {

        Student obj = new Student( age: 20,  rollNumber: 76,  name: "Ram");

        //creating Clone of obj
        Student cloneObj = new Student();
        cloneObj.name = obj.name;
        cloneObj.age = obj.age;
        cloneObj.rollNumber = obj.rollNumber;
    }
}
```

// original obj

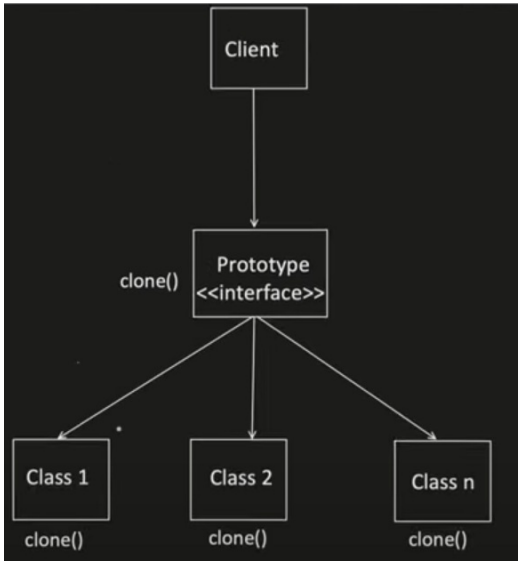
clone obj

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.

```java
public class Main {

    public static void main(String args[]){

        Student obj = new Student( age: 20,  rollNumber: 75,  name: "Ram");
        Student cloneObj = (Student) obj.clone();
    }
}
```

```java
public interface Prototype {

    Prototype clone();
}
```

```java
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);
    }
}
```

⇒ 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 Intitalization (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;
    }
}
```

```
public class Main {

    public static void main(String args[]) {

      DBConnection connObject = DBConnection.getInstance();
    }

}
```

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

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

→ Lazy Initialization

null initially

```
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

    public static DBConnection getInstance(){

        if(conObject == null){
            conObject = new DBConnection();
        }
        return conObject;
    }
}
```

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 locking (used in industry)

```java
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.

⇒ Issue in double locking

issue1 Reordering of Instructions :- CPU automatically does this, which can cause issue.

issue2 L1 caching :- Caching storage by multiple threads can cause 1 variable to be intialised with default value instead of what we wanted to assign.

⇒ Solution :- volatile keyword for singleton object

```
public class DBConnection {

    private static volatile DBConnection conObject;
    int memberVariable;

    private DBConnection(int memberVariableValue){
        this.memberVariable = memberVariableValue;
    }

    public static DBConnection getInstance(){

        if(conObject == null){

            synchronized(DBConnection.class){

                if(conObject == null){

                    conObject = new DBConnection( memberVariableValue: 10);
                }
            }
        }
        return conObject;
    }
}
```

same code with just 1 extra keyword

how it resolves :-

property1 :- it read/write directly from memory. so now data is not storing in caching so L1 caching is resolved.

property2 :- commands can't reorder through it. means

- statement 1
- statement 2
- statement 3
- volatile obj.
- statement 4
- statement 5
- statement 6

st1 & st4 can't reorder only 1,2,3 can & 4,5,6 can seperately.

⇒ Flyweight design pattern (this pattern saves memory by sharing data among diff. obj.)

This pattern is a way to save memory in applications that creates a large no. of similar objects.
So instead of creating new object everytime, flyweight uses same obj. again.

Imp Ques
1) Design word processor
2) Design Game

Que) Gaming scenario :- we have many humanoid robots (5 lakh) and many dog robots (5 lakh) and all shares some similar data.

```java
public class Robot {

    int coordinateX;
    int coordinateY;
    String type;
    Sprites body; //small 2d bitmap (graphic element)

    Robot(int x, int y, String type, Sprites body){

        this.coordinateX = x;
        this.coordinateY = y;
        this.type = type;
        this.body = body;
    }

    //getter and setters
}
```

```java
public class Sprites {
}
```

Let's just imagine that sprites is an animation or image of player (quite heavy).

Problem

```java
public class Main {

    public static void main(String args[]){

        int x=0;
        int y=0;
        for(int i=0; i<500000; i++){

            Sprites humanoidSprite = new Sprites();
            Robot humanoidRobotObject = new Robot( x: x+i,  y: y+i,  type: "HUMANOID", humanoidSprite);
        }

        for(int i=0; i<500000; i++){
            Sprites roboticDogSprite = new Sprites();
            Robot roboticDogObject = new Robot( x: x+i,  y: y+i,  type: "ROBOTIC_DOGS", roboticDogSprite);
        }
    }
}
```

assume each obj. is 30 Kb and we create 10 Lakh obj. then game will be of 30 GB of size. (too big)

Note:- all obj. shares some similar info, just present in diff. co-ordinates.

So, when to use this pattern

- When Memory is Limited.
- When Objects shared data. ★
  - Intrinsic data : shared among objects and remain same once defined one value.
  - Extrinsic data : changes based on client input and differs from one object to another.
- Creation of Object is expensive.

Note:- Like in above example, Intrisic data is HUMANOID & humanSprite & extrinsic data is co-ordinates

⇒ How to resolve the issue (Steps)

- From Object, remove all the Extrinsic data and keep Intrinsic Data(this object called Flyweight Object)
- This Flyweight Class can be immutable.
- Extrinsic Data can be passed to the Flyweight class in method parameter.
- Once the Flyweight Object is created, it is Cached and reused whenever required.

means make fields private and also only provide getters & no setters.

private

```java
public interface IRobot {

    public void display(int x, int y);
}
```

kept only intrinsic data

```java
public class HumanoidRobot implements IRobot {

    private String type;
    private Sprites body; //small 2d bitmap (graphic element)

    HumanoidRobot(String type, Sprites body){
        this.type = type;
        this.body = body;
    }

    public String getType() {
        return type;
    }

    public Sprites getBody() {
        return body;
    }

    @Override
    public void display(int x, int y) {

        //use the humanoid sprites object
        // and X and Y coordinate to render the image.

    }
}
```

only getters

extrinsic data is passed

```java
public class RoboticDog implements IRobot{

    private String type;
    private Sprites body; //small 2d bitmap (graphic eleme

    RoboticDog(String type, Sprites body){
        this.type = type;
        this.body = body;
    }

    public String getType() { return type; }

    public Sprites getBody() { return body; }

    @Override
    public void display(int x, int y) {

        //use the Robotic Dog sprites object
        // and X and Y coordinate to render the image.

    }
}
```

3

Flyweight object (used for caching)

```java
public class RoboticFactory {

    private static Map<String, IRobot> roboticObjectCache = new HashMap<>();

    public static IRobot createRobot(String robotType){

        if(roboticObjectCache.containsKey(robotType)){
            return roboticObjectCache.get(robotType);
        }
        else {
            if(robotType == "HUMANOID"){
                Sprites humanoidSprite = new Sprites();
                IRobot humanoidObject = new HumanoidRobot(robotType, humanoidSprite);
                roboticObjectCache.put(robotType, humanoidObject);
                return humanoidObject;
            }
            else if(robotType == "ROBOTICDOG"){
                Sprites roboticDogSprite = new Sprites();
                IRobot roboticDogObject = new RoboticDog(robotType, roboticDogSprite);
                roboticObjectCache.put(robotType, roboticDogObject);
                return roboticDogObject;
            }
        }
        return null;
    }
}
```

cached memory

Key = HUMANOID or ROBOTIC_DOG | val is object

first check if already present & create if not present

```java
public class Sprites {
}
```

Client code

```java
public class Main {

    public static void main(String args[]){

        IRobot humanoidRobot1 = RoboticFactory.createRobot( robotType: "HUMANOID");
        humanoidRobot1.display( x: 1, y: 2);


        IRobot humanoidRobot2 = RoboticFactory.createRobot( robotType: "HUMANOID");
        humanoidRobot2.display( x: 10, y: 30);

        IRobot roboDog1 = RoboticFactory.createRobot( robotType: "ROBOTICDOG");
        roboDog1.display( x: 2, y: 9);

        IRobot roboDog2 = RoboticFactory.createRobot( robotType: "ROBOTICDOG");
        roboDog2.display( x: 11, y: 19);

    }
}
```

only the coordinates are changing

Ques Text processor scenario :- we have to impliment a text editor like notepad

(we obviously type same character many times, so we don't have to create obj of each character every time)

Problem :

```java
public class Character {

    char character;
    String fontType;
    int size;
    int row;
    int column;

    Character(char character, String fontType, int size, int row, int column){
        this.character = character;
        this.fontType = fontType;
        this.size = size;
        this.row = row;
        this.column = column;
    }

    //getter and setters
}
```

Intrinsic data

extrinsic data

```java
public class Main {

    public static void main(String args[]){

        /*
            This is the data we want to write into the word processor.

            Total = 58 characters
            t = 7 times
            h = 3 times
            a = 3 times and so on...

         */

        Character object1 = new Character( character: 't', fontType: "Arial", size: 10, row: 0, column: 0);
        Character object2 = new Character( character: 'h', fontType: "Arial", size: 10, row: 0, column: 1 );
        Character object3 = new Character( character: 'i', fontType: "Arial", size: 10, row: 0, column: 2 );
        Character object4 = new Character( character: 's', fontType: "Arial", size: 10, row: 0, column: 3 );
    }
}
```

→ will become a memory issue

Solution

```java
public interface ILetter {

    public void display(int row, int column);
}
```

```java
public class DocumentCharacter implements ILetter{

    private char character;
    private String fontType;
    private int size;

    DocumentCharacter(char character, String fontType, int size){
        this.character = character;
        this.fontType = fontType;
        this.size = size;

    }

    //only getter methods

    @Override
    public void display(int row, int column) {

        //display the character of particular font and size
        //at given location
    }
}
```

private

```java
public class LetterFactor {

    private static Map<Character, ILetter> characterCache = new HashMap<>();

    public static ILetter crateLetter(char characterValue){

        if(characterCache.containsKey(characterValue)){
            return characterCache.get(characterValue);
        }
        else {

            DocumentCharacter characterObj = new DocumentCharacter(characterValue,  fontType: "Arial",  size: 10);
            characterCache.put(characterValue, characterObj);
            return characterObj;
        }
    }
}
```

```java
public class Main {

    public static void main(String args[]){

        /*
            this is the data we want to write into the word processor

            Total = 58 characters
            t = 7 times
            h = 3 times
            a = 3 times and so on...

         */

        ILetter object1 = LetterFactor.crateLetter( characterValue: 't');
        object1.display( row: 0, column: 0);

        ILetter object2 = LetterFactor.crateLetter( characterValue: 't');
        object1.display( row: 0, column: 6);

    }
}
```

⇒ Command design pattern (behavioural design)

↳ let's take we case of remote control which give command to control various home-appliances (ac and tv)

Imp Question :- how do we implement undo and redo functionality

Ex:- Simple example

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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();
    }
```

process

```
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

problems :-

- Lack of Abstraction : → means client should not be responsible for implementing the process
Today, process of turning on AC is simple, but if there are more steps, client has to aware all of that, which is not good.

- Undo/Redo Functionality: → who will do undo/redo
What if I want to implement the undo/redo capability. How it will be handled.

- Difficulty in Code Maintenance:
What if in future, we have to support more commands for more devices example Bulb. Ma

Note:- we can't tell client to perform undo/redo and AirCond^{n} is just a dumb object.

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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();

        Bulb bulbObj = new Bulb();
        bulbObj.turnOnBulb();
        bulbObj.turnOffBulb();
    }
}
```

what if bulb is also present in future then does client need to know functionlity of all.?

No

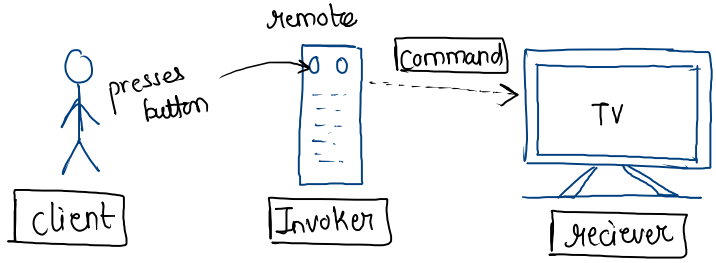
Very tightly coupled

```java
public class Bulb {

    boolean isOn;

    public void turnOnBulb(){
        isOn = true;
        System.out.println("Bulb is ON");
    }

    public void turnOffBulb(){
        isOn = false;
        System.out.println("Bulb is OFF");
    }
}
```

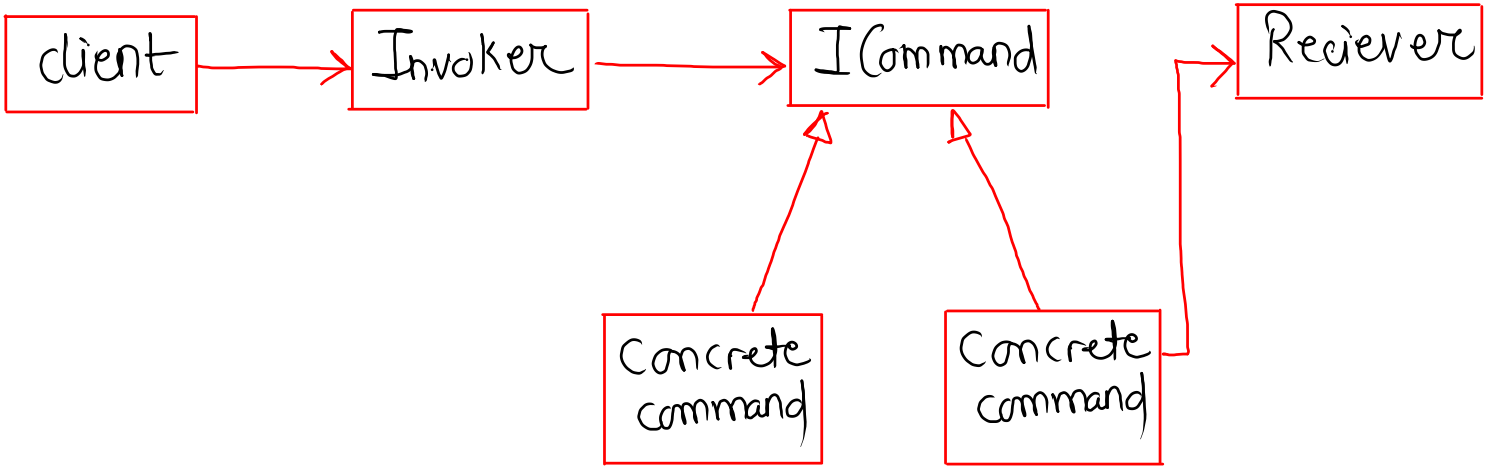
```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF")
    }

    public void setTemperature(int temp
        this.temperature = temp;
        System.out.println("Temperature
    }
}
```

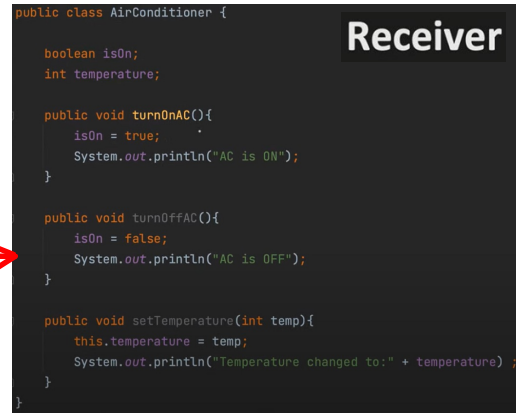
→ So, Command design pattern saperates the logic in 3 parts.

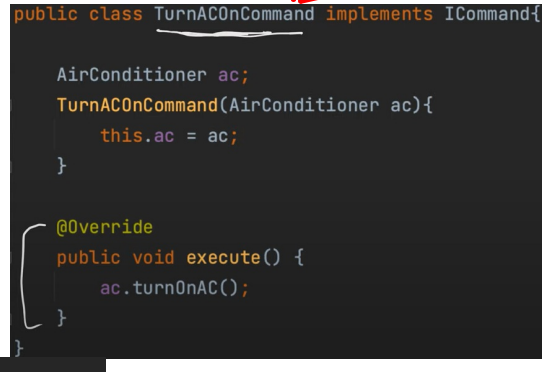
- Reciever
- Invoker
- Command

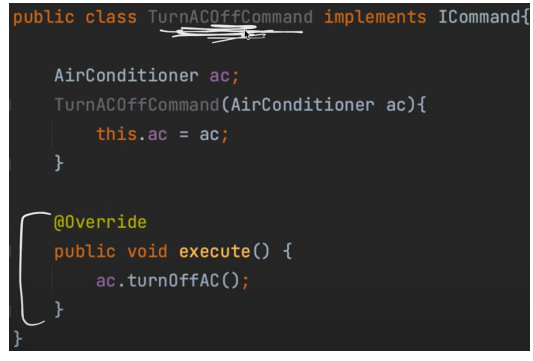


UML









Note:- now, here lack of abstraction and code maintainance is also resolved but how do we implement undo/redo functionality.

```java
import java.util.Stack;

public class MyRemoteControl {

    Stack<ICommand> acCommandHistory = new Stack<>();
    ICommand command;

    MyRemoteControl(){
    }

    public void setCommand(ICommand command){
        this.command = command;
    }

    public void pressButton()
    {
        command.execute();
        acCommandHistory.add(command);
    }

    public void undo(){

        if(!acCommandHistory.isEmpty()){
            ICommand lastCommand = acCommandHistory.pop();
            lastCommand.undo();
        }
    }
}
```

(Invoker)

save command in stack

take out the last command from stack

(reciever is same)

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

(Command)

```java
public interface ICommand {

    public void execute();
    public void undo();
}
```

additional functionality

client

(same)

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

        //AC object
        AirConditioner airConditioner = new AirConditioner();

        //remote
        MyRemoteControl remoteObj = new MyRemoteControl();

        //create the command and press the button
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();

        //undo the last operation
        remoteObj.undo();
    }
}
```

extra

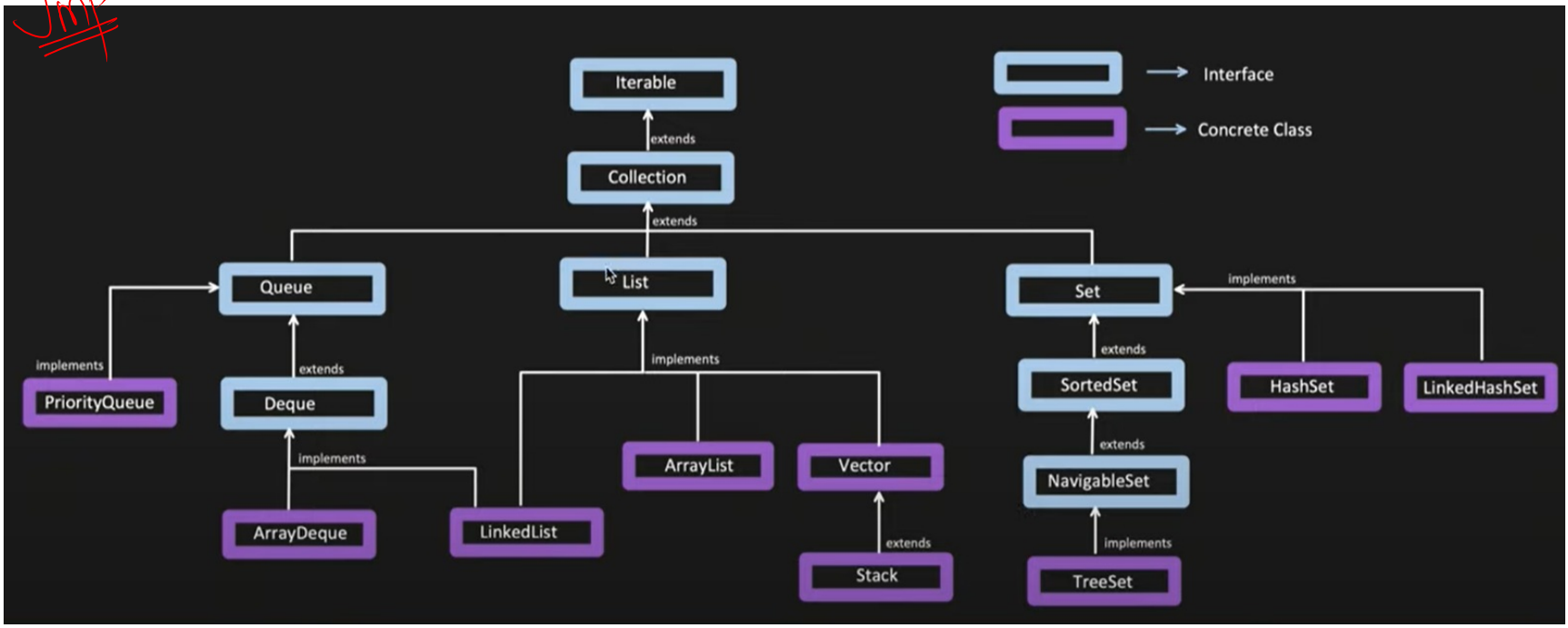
```java
public class TurnACOnCommand implements ICommand{

    AirConditioner ac;
    TurnACOnCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOnAC();
    }

    @Override
    public void undo() {
        ac.turnOffAC();
    }
}
```

simply opposite of execute

```java
public class TurnACOffCommand implements ICommand{

    AirConditioner ac;
    TurnACOffCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOffAC();
    }

    @Override
    public void undo() {
        ac.turnOnAC();
    }
}
```

⇒ Iterator design pattern (best example is java collections itself)

Imp



Ex:-

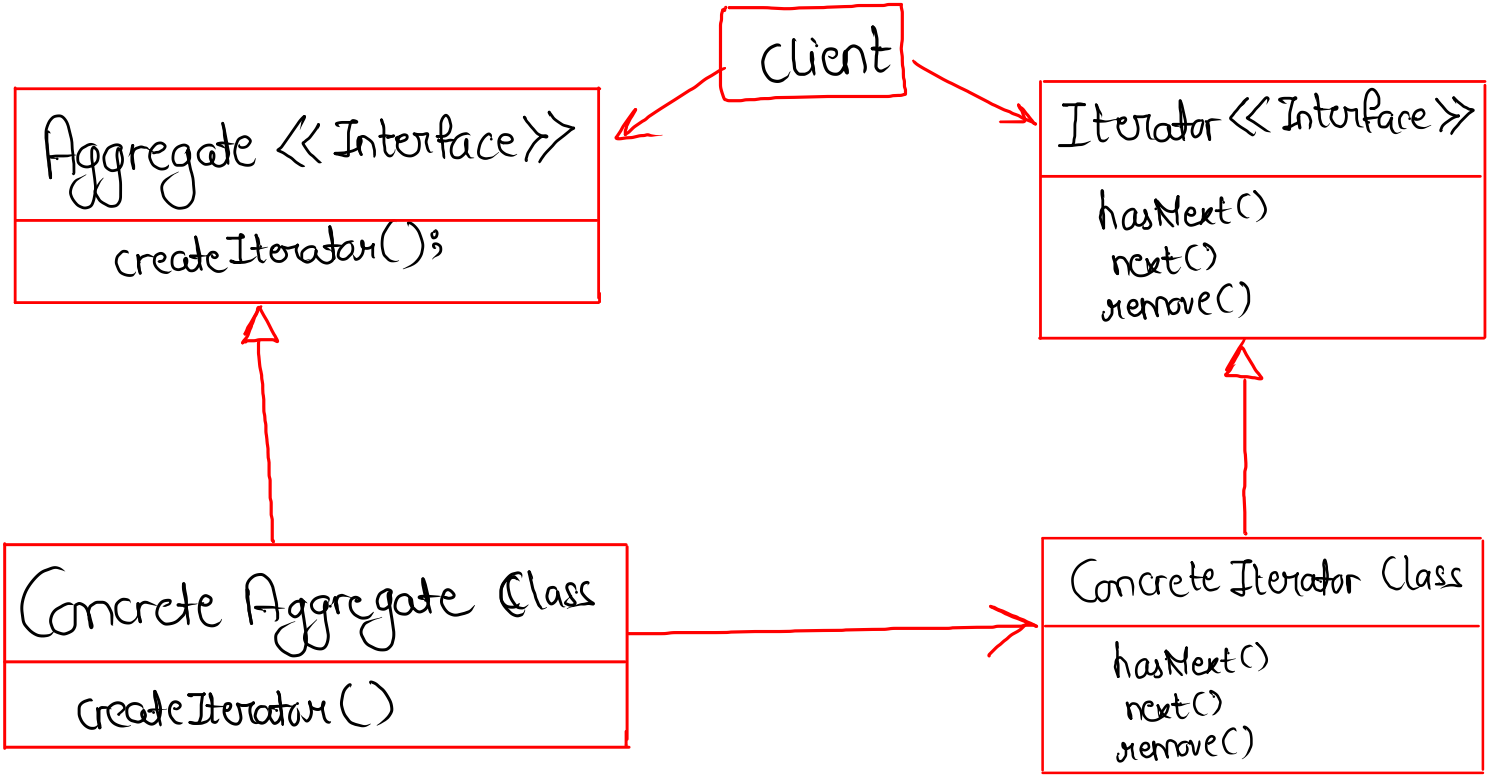
```
Iterator<Integer> itr = map.iterator();
while (itr.hasNext()){
    int val = itr.next();
    Syso(val);
}
```

Note:- here, we do not need to worry that map is what collection (Hashmap or set or stack or Vector or etc).
we can directly use iterator.

define

So, its a Behavioral Design pattern, that provides a way to access element of a Collection sequentially without exposing the underlying representation of the collection.

UML



we can have many impliment" of aggregate & iterator as well and each aggregate will be using (has-a) of any 1 of the iterator implementation.

Ex:-

```java
public interface Aggregate {

    Iterator createIterator();
}
```

```java
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

```java
public class Library {

    private List<Book> booksList;

    public Library(List<Book> booksList) {
        this.booksList = booksList;
    }

    public Iterator createIterator() {
        return new BookIterator(booksList);
    }
}
```

each aggregate has to use its own iterator

```java
public class BookIterator implements Iterator {
    private List<Book> books;
    private int index = 0;

    public BookIterator(List<Book> books) {
        this.books = books;
    }

    @Override
    public boolean hasNext() {
        return index < books.size();
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return books.get(index++);
        }
        return null;
    }
}
```

```java
public class Client {

    public static void main(String[] args) {
        List<Book> booksList = Arrays.asList(
                new Book( price: 100,  bookName: "Science"),
                new Book( price: 200,  bookName: "Maths"),
                new Book( price: 300,  bookName: "GK"),
                new Book( price: 400,  bookName: "Drawing")
        );

        Library lib = new Library(booksList);
        Iterator iterator = lib.createIterator();

        while (iterator.hasNext()) {
            Book book = (Book) iterator.next();
            System.out.println(book.getBookName());
        }
    }
}
```

```java
public class Book {

    private int price;
    private String bookName;

    Book(int price, String bookName){
        this.price = price;
        this.bookName = bookName;
    }
    public int getPrice() {
        return price;
    }

    public String getBookName() {
        return bookName;
    }
}
```

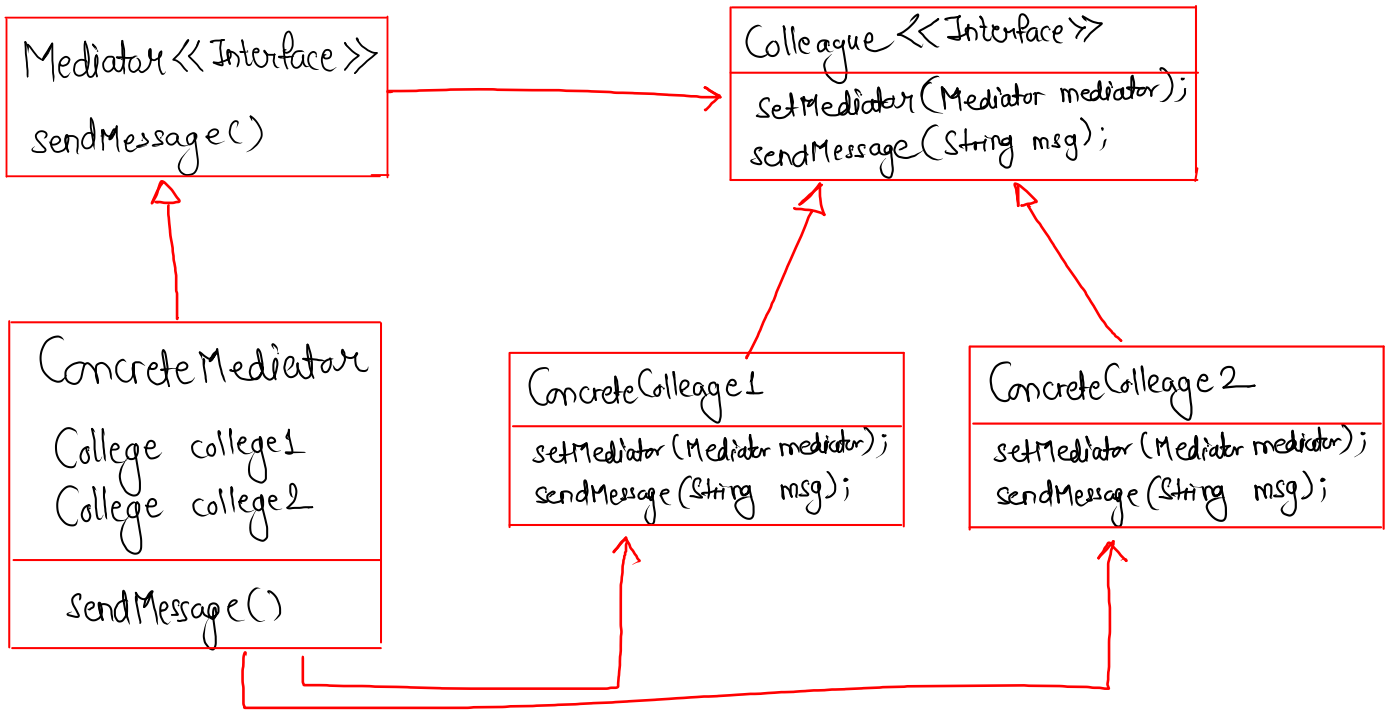
⇒ Mediator design pattern (very similar to proxy)

↳ famous ex:- online auction app, airline app.

- The Mediator Pattern is a behavioral design pattern.
- It encourage loose coupling by keeping objects from referring to each other explicitly and allows them to ***communicate through a mediator*** object.

} means whenever 2 objects wants to talk to each other then instead of directly talking they will talk with help of an mediator

UML



Ex:-

```java
//this is Mediator Interface
public interface AuctionMediator {
    void addBidder(Colleague bidder);
    void placeBid(Colleague bidder, int bidAmount);
}
```

```java
public interface Colleague {

    void placeBid(int bidAmount);
    void receiveBidNotification(int bidAmount);
    String getName();
}
```

```java
//Mediator Concrete Class
public class Auction implements AuctionMediator {

    List<Colleague> colleagues = new ArrayList<>();

    @Override
    public void addBidder(Colleague bidder) {
        colleagues.add(bidder);
    }

    @Override
    public void placeBid(Colleague bidder, int bidAmount) {

        for(Colleague colleague : colleagues){
            if(!colleague.getName().equals(bidder.getName())){
```

```java
public class Bidder implements Colleague {

    String name;
    AuctionMediator auctionMediator;

    Bidder(String name, AuctionMediator auctionMediator) {
        this.name = name;
        this.auctionMediator = auctionMediator;
        auctionMediator.addBidder(this);
    }

    @Override
    public void placeBid(int bidAmount) {
        auctionMediator.placeBid( bidder: this, bidAmount);
    }

    @Override
    public void receiveBidNotification(int bidAmount) {
        System.out.println("Bidder: " + name + " got the notification that someone has put bid of : " + bidAmount);
    }

    @Override
    public String getName(){
        return name;
    }
}
```

```java
public class Main {

    public static void main(String args[]){

        AuctionMediator auctionMediatorObj = new Auction();
        Colleague bidder1 = new Bidder( name: "A", auctionMediatorObj);
        Colleague bidder2 = new Bidder( name: "B", auctionMediatorObj);

        bidder1.placeBid( bidAmount: 2000);
        bidder2.placeBid( bidAmount: 3000);
        bidder1.placeBid( bidAmount: 3001);
    }
}
```

⇒ Visitor design pattern (Hotel booking system)

Whats the problem with the below class?

```
public class HotelRoom {

    public void getRoomPrice(){
        //price computation logic
    }

    public void initiateRoomMaintenance(){
        //start room maintenance
    }

    public void reserveRoom(){
        //perform operation to reserve the room
    }

    //many more operations can come over the time
}
```

Problems :-

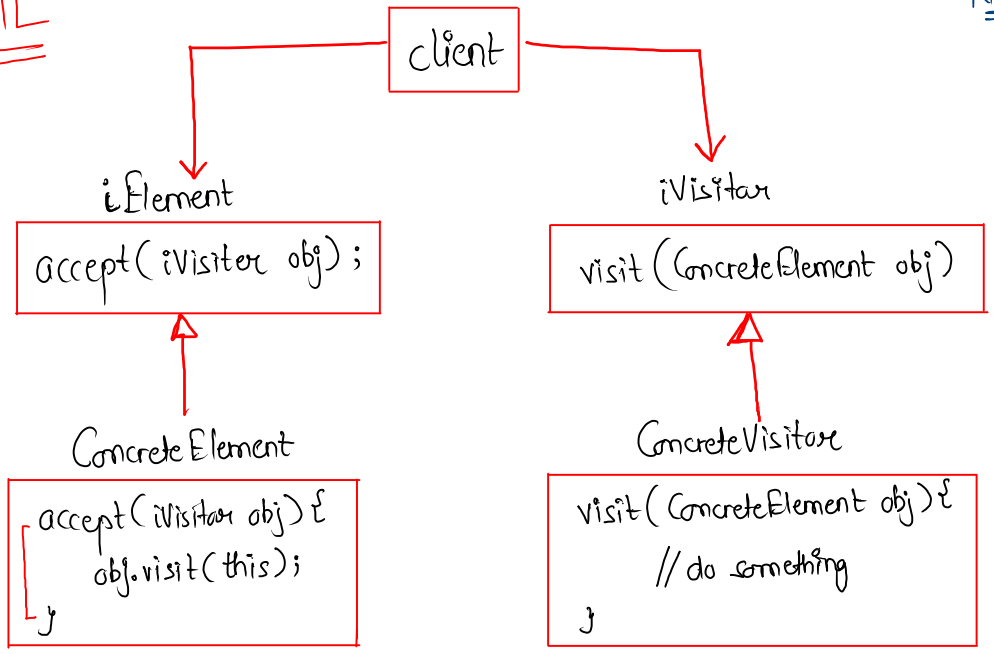
1) if other functionalities are added in future then we have to test the entire class again.

2) if let say 50 more functionalities added in future then this class can go huge

Note :- visitor design pattern saperates the functionality from class (we will create saperate classes for saperate functionalities)

- It is a **Behavioral design pattern**
- That allows you to add new operations to existing classes without changing their structure.
- It achieves this by separating the algorithm from the objects on which it operates.
- It does **Double Dispatch** to achieve this.
  (Double Dispatch means, method which need to be invoked decided by the caller object and the object passed in the argument.)

UML



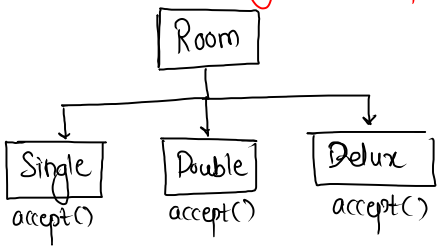
Note:- iElement or iVisitor means interface element or interface visitar

advantage is:-

1) when any new operation added then we dont have to test the element class.
2) will any one particular visitar class grow? no, because there are fixed no. of methods in visitar interface
   only more visitar class can be added for different functionalities.
   (means visitar classes can grow horizontaly and not vertically)


Kunal Suri
public class Client {
public static void main(String args[]){
RoomElement singleRoomObj = new SingleRoom();
RoomElement doubleRoomObj = new DoubleRoom();
RoomElement deluxeRoomObj = new DeluxeRoom();
//performing an operation on the objects
RoomVisitor pricingVisitorObj = new RoomPricingVisitor();
singleRoomObj.accept(pricingVisitorObj);
System.out.println(((SingleRoom)singleRoomObj).roomPrice);
doubleRoomObj.accept(pricingVisitorObj);
System.out.println(((DoubleRoom)doubleRoomObj).roomPrice);
deluxeRoomObj.accept(pricingVisitorObj);
System.out.println(((DeluxeRoom)deluxeRoomObj).roomPrice);
//performing another operation on the objects
RoomVisitor maintenanceVisitorObj = new RoomMaintenanceVisitor();
singleRoomObj.accept(maintenanceVisitorObj);
doubleRoomObj.accept(maintenanceVisitorObj);
deluxeRoomObj.accept(maintenanceVisitorObj);
}
}
public interface RoomElement {
public void accept(RoomVisitor visitor);
}
public class SingleRoom implements RoomElement{
public int roomPrice = 0;
@Override
public void accept(RoomVisitor visitor) {
visitor.visit(singleRoomObj: this);
}
}
means SingleRoom object
(element)
public class DoubleRoom implements RoomElement{
public int roomPrice = 0;
@Override
public void accept(RoomVisitor visitor) {
visitor.visit(doubleRoomObj: this);
}
}
means doubleRoom obj.
(element)
public class DeluxeRoom implements RoomElement{
public int roomPrice = 0;
@Override
public void accept(RoomVisitor visitor) {
visitor.visit(deluxeRoomObj: this);
}
}
means DeluxRoom obj.
(element)
public interface RoomVisitor {
public void visit(SingleRoom singleRoomObj);
public void visit(DoubleRoom doubleRoomObj);
public void visit(DeluxeRoom deluxeRoomObj);
}
public class RoomPricingVisitor implements RoomVisitor{
@Override
public void visit(SingleRoom singleRoomObj) {
System.out.println("Pricing computation logic of SingleRoom");
singleRoomObj.roomPrice = 1000;
}
@Override
public void visit(DoubleRoom doubleRoomObj) {
System.out.println("Pricing computation logic of DoubleRoom");
doubleRoomObj.roomPrice = 2000;
}
@Override
public void visit(DeluxeRoom deluxeRoomObj) {
System.out.println("Pricing computation logic of DeluxeRoom");
deluxeRoomObj.roomPrice = 5000;
}
}
(visitor)
public class RoomMaintenanceVisitor implements RoomVisitor{
@Override
public void visit(SingleRoom singleRoomObj) {
System.out.println("Performing maintenance of SingleRoom");
}
@Override
public void visit(DoubleRoom doubleRoomObj) {
System.out.println("Performing maintenance of DoubleRoom");
}
@Override
public void visit(DeluxeRoom deluxeRoomObj) {
System.out.println("Performing maintenance of DeluxeRoom");
}
}
(visitor)
Note:- each visitor is handling one functionality and we can add as many visitors we want & now we do not have to test element class when 1 funct" is added (only test visitor class).

Ques What is single dispatch ?




```
Room obj = new Delux();
obj.accept();
```

caller

which accept method to call ? depend upon 1 object only
this is called single dispatch.

Ques What is double dispatch ?

when deciding which accept method to call depend upon 2 objects (caller and argument)

Interfaces

```
RoomElement obj = new SingleRoom();
RoomVisitor pricingobj = new RoomPricingVisitor();
obj.accept(pricingobj);
```

caller

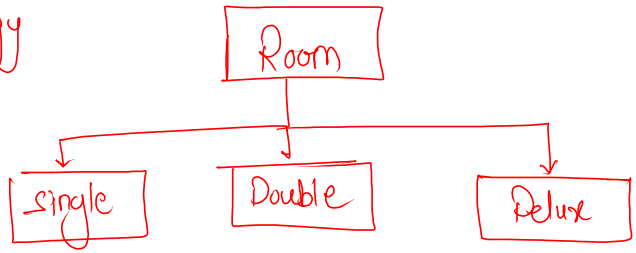
another caller

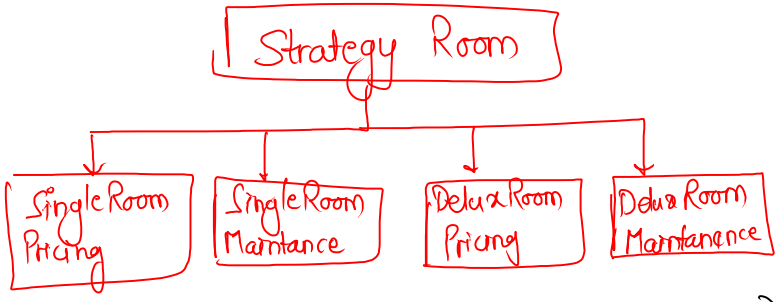
accept method of single room is called (single dispatch) which is again calling visit method in which we are passing this keyword

argument is telling now under RoomPricingVisitor class which visit method to call, it depends upon the this keyword (double dispatch)

Note:- do not get confuse b/w strategy & visitor design pattern.

Strategy

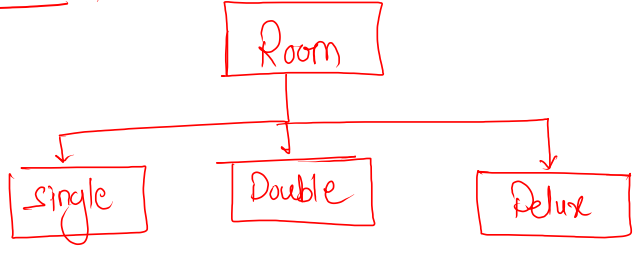


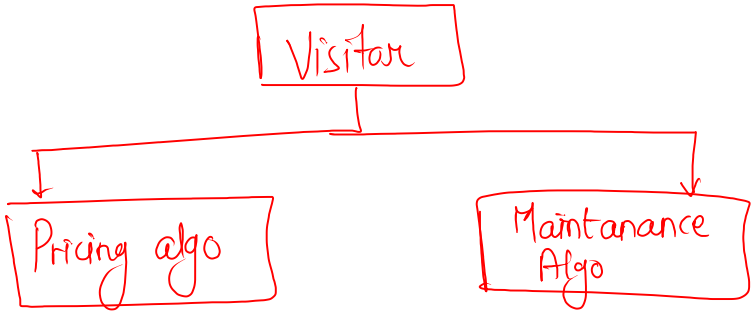


(here class are not independent of element)

In Visitor, each visitor class is independent of element

Visitor,





now, each element can decide independently that which algo he want to use

⇒ Memento design pattern / Snapshot design pattern

Why its required and When to use:
Provides an ability to revert the an object to a previous state i.e. UNDO capability.
And
It does not expose the object internal implementation.

Pattern Category: It's a behavioral pattern.

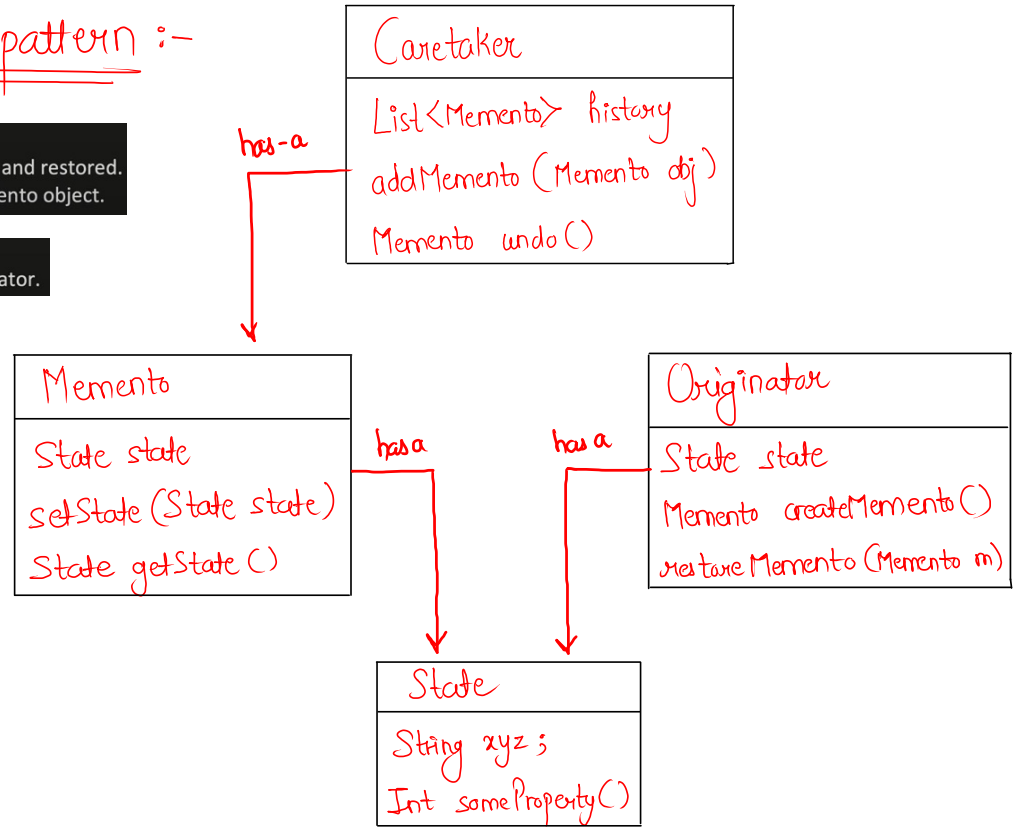
Major components of memento pattern :-

Originator:
- It represents the object, for which state need to be saved and restored.
- Expose Methods to Save and Restore its state using Memento object.

Memento:
- It represents an Object which holds the state of the Originator.

Caretaker:
- Manages the list of States (i.e. list of Memento)

UML :-



```java
public class ConfigurationCareTaker {

    List<ConfigurationMemento> history = new ArrayList<>();

    public void addMemento(ConfigurationMemento memento) {
        history.add(memento);
    }

    public ConfigurationMemento undo() {
        if (!history.isEmpty()) {
            int lastMementoIndex = history.size() - 1;
            //get the last memento from the list
            ConfigurationMemento lastMemento = history.get(lastMementoIndex);
            //remove the last memento from the list now
            history.remove(lastMementoIndex);
            return lastMemento;
        }
        return null;
    }
}
```

has-a

has-a

```java
public class Client {

    public static void main(String args[]){

        ConfigurationCareTaker careTakerObject = new ConfigurationCareTaker();
        //initiate State of the originator
        ConfigurationOriginator originatorObject = new ConfigurationOriginator( height: 5,  width: 10);

        //save it
        ConfigurationMemento snapshot1 = originatorObject.createMemento();

        //add it to history
        careTakerObject.addMemento(snapshot1);

        //originator changing to new state
        originatorObject.setHeight(7);
        originatorObject.setWidth(12);

        //save it
        ConfigurationMemento snapshot2 = originatorObject.createMemento();

        //add it to history
        careTakerObject.addMemento(snapshot2);
        //originator changing to new state
        originatorObject.setHeight(9);
        originatorObject.setWidth(14);

        //UNDO
        ConfigurationMemento restoredStateMementoObj = careTakerObject.undo();
        originatorObject.restoreMemento(restoredStateMementoObj);

        System.out.println("height: " + originatorObject.height + " width: " + originatorObject.width);

    }
}
```

has-a

has-a

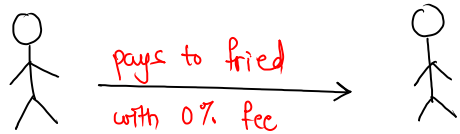
```java
//Memento
public class ConfigurationMemento {

    int height;
    int width;

    public ConfigurationMemento(int height, int width){
        this.height = height;
        this.width = width;
    }

    public int getHeight() {
        return height;
    }
    public int getWidth() {
        return width;
    }
}
```

```java
//Originator
public class ConfigurationOriginator {

    int height;
    int width;

     ConfigurationOriginator(int height, int width){
        this.height = height;
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public void setWidth(int width) {
        this.width = width;
    }
    public ConfigurationMemento createMemento(){
        return new ConfigurationMemento(this.height, this.width);
    }

    public void restoreMemento(ConfigurationMemento mementoToBeRestored){
        this.height = mementoToBeRestored.height;
        this.width = mementoToBeRestored.width;
    }
}
```

⇒ Template Method design pattern

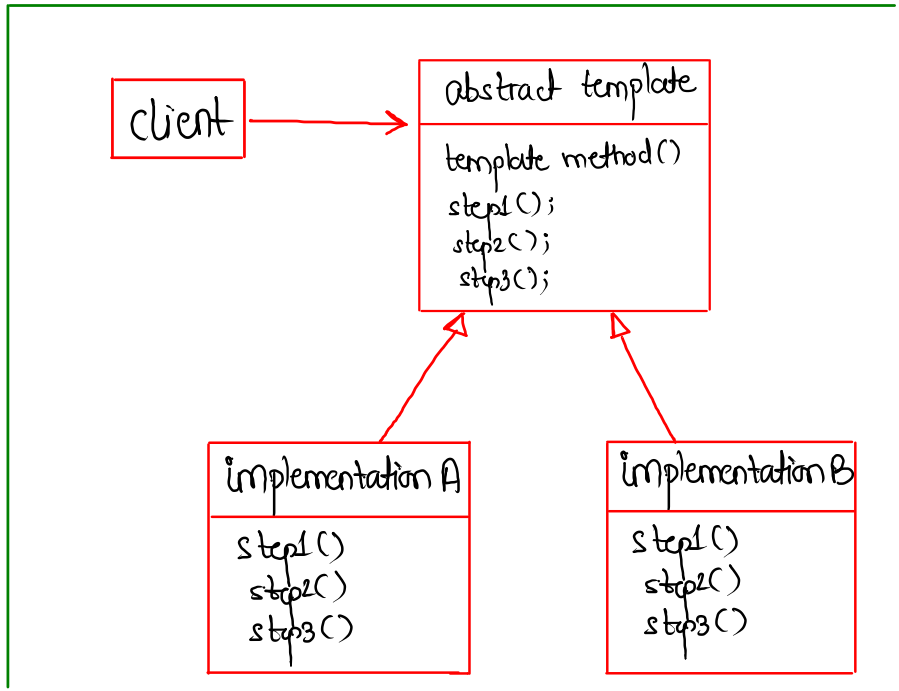
- It is a behavioural design pattern
- When to use it :- when we want all class to follow some specific steps to process the task but also need to provide the flexibility that each class can have their own logic in that specific step.

Ex:- Payment gateway



Specific steps:-

1) validate request
2) Debit money
3) calculate fees (0% for fried & 2% for merchant)
4) Credit money



client code

```
PaymentFlow obj = new PayToFriend();
obj.sendMoney();
```

```java
public abstract class PaymentFlow {

    public abstract void validateRequest();
    public abstract void calculateFees();
    public abstract void debitAmount();
    public abstract void creditAmount();

    //this is Template method: which defines the order of steps to execute the task.
    public final void sendMoney(){
        //step1
        validateRequest();

        //step2
        debitAmount();

        //step3
        calculateFees();

        //step4
        creditAmount();
    }
}
```

template is method, that is why it is template METHOD
design pattern & this method is final so no child can override this method.

```java
public class PayToFriend extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToFriend");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToFriend");
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("0% fees charged");
    }
    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the full amount");
    }
}
```

```java
public class PayToMerchantFlow extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToMerchantFlow");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToMerchantFlo
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("2% fees charged");
    }
    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the remaining amount");
    }
```

⇒ Interpreter design pattern

↳ It is a behavioural design pattern

↳ It is specifically used to evaluate mathematic expressions.

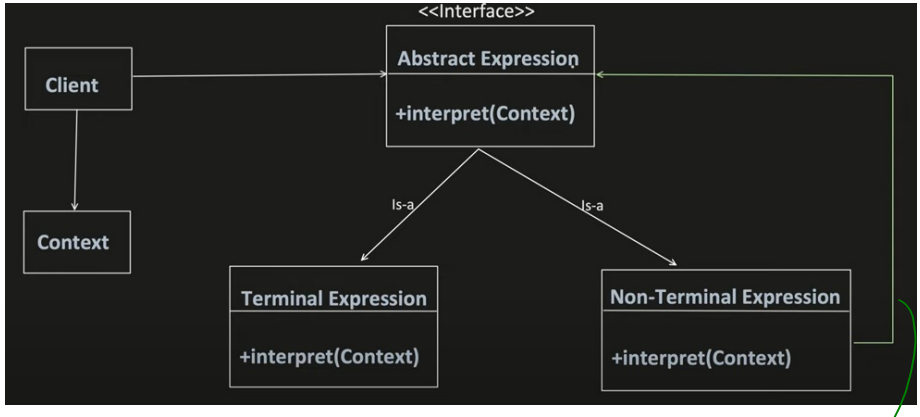
Components Involved

- Context
- Abstract Expression
  - Terminal Expression
  - Non-terminal expression

Expression:- $(a * b) + (c * d)$

- $(a * b)$ → terminal (which can't be further divided)
- $(c * d)$ → non-terminal (which can be further divided into terminal or non-terminal expressions)

UML :-



Recursive statement

```java
public class Client {

    public static void main(String args[]) {

        //initialize the context
        Context context = new Context();
        context.put( strVariable: "a",  intValue: 2);
        context.put( strVariable: "b",  intValue: 4);
        context.put( strVariable: "c",  intValue: 8);
        context.put( strVariable: "d",  intValue: 16);

        // ((a *b) + (c*d))
        AbstractExpression expression2 = new BinaryNonTerminalExpression(
                new BinaryNonTerminalExpression(
                        new NumberTerminalExpression( stringVal: "a"), new NumberTerminalExpression( stringVal: "b"),  operator: '*'),
                        new BinaryNonTerminalExpression(
                                new NumberTerminalExpression( stringVal: "c"), new NumberTerminalExpression( stringVal: "d"),  operator: '*'),
                 operator: '+');
        System.out.println(expression2.interpret(context));
    }
}
```

has-a

has-a

```java
public class Context {

    Map<String, Integer> contextMap = new HashMap<>();

    public void put(String strVariable, int intValue) {
        contextMap.put(strVariable, intValue);
    }

    public int get(String strVariable) {
        return contextMap.get(strVariable);
    }
}
```

```java
public interface AbstractExpression {

    int interpret(Context context);
}
```

has-a

is-a

is-a

```java
public class NumberTerminalExpression implements AbstractExpression{

    String stringValue;

    NumberTerminalExpression(String stringVal) {
        this.stringValue = stringVal;
    }

    @Override
    public int interpret(Context context) {
        return context.get(stringValue);
    }
}
```

```java
public class BinaryNonTerminalExpression implements AbstractExpression{

    AbstractExpression leftExpression;
    AbstractExpression rightExpression;
    char operator;

    public BinaryNonTerminalExpression(AbstractExpression leftExpression, AbstractExpression rightExpressi
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
        this.operator = operator;
    }

    @Override
    public int interpret(Context context) {
        switch (operator) {
            case '+':
                return leftExpression.interpret(context) + rightExpression.interpret(context);
            case '*':
                return leftExpression.interpret(context) * rightExpression.interpret(context);
            default:
                return 0;
        }
    }
```