UML Diagram

Date: / /
Page No.:

italics:

→ <<interface>>
Name
method()

Interface : classes implements interfaces, denoted by generalization.

→ An Abstract class is identified by specifying its name is italics.

→ ClassName
Property_name : type
method() : type

ClassName →

ClassName
+ type : Type
− VehicleId : int
VehicleNum : String
+ getVehicleParts(int id) : list<parts>
− changeType(Type type) : Type
#

Attribute

underline means static

+ ← Public attributes/method
− ← Private attributes/method
← Protected attributes/method

mtd Name

mtd Name
argument
return type
accessibility
type

_ ← underline means an attributes or method is static

⇒ **Association** :- A relationship where one class is aware of another, allowing it to intract,

types - Ex:- Room & Person

Person → Room

A can call B

B can call A

Ex:- train & station (i) Unidirectional Association

A —— B

(ii) Bidirectional Association

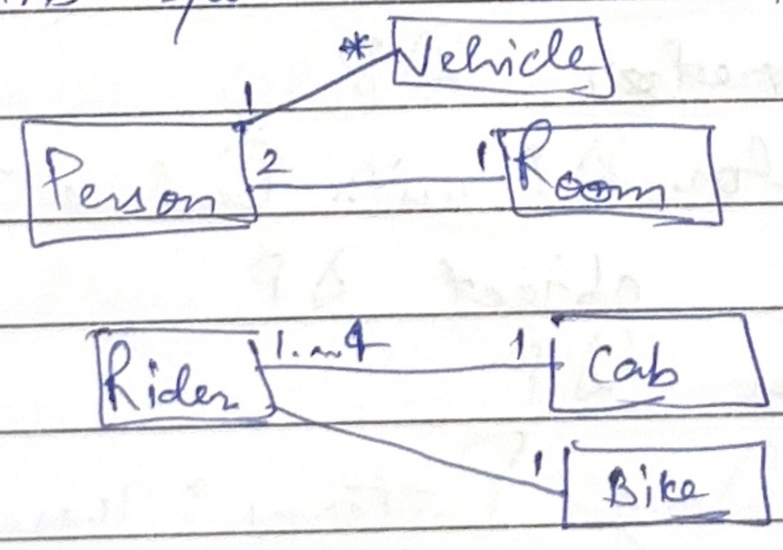
A ——→ B

A can call B but B can not call A.

* **multiplicity** :-

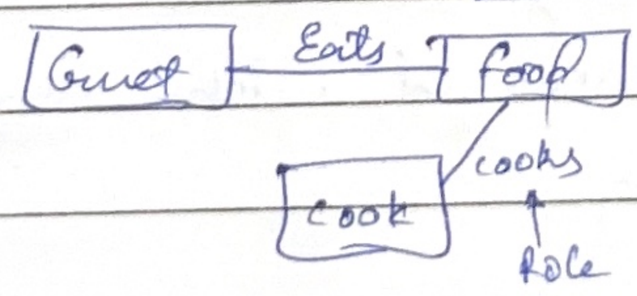
Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities b/w two classes. for example.

Ex:

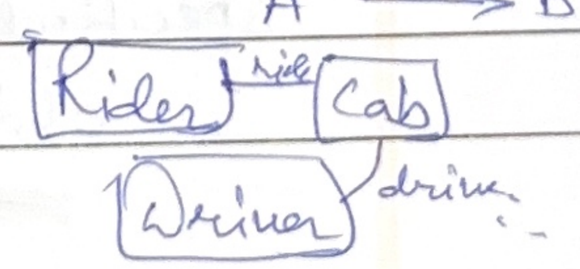


* ← Any number.
n ← Exact number
1....m ← min value 1
max value m

* **Role** :-



A —uses→ B



Date: / /
Page No.:

* **Aggregation:-** "has-an" relationship. "weaker relationship"

Aggregation implies a relationship where the child can exist independently of the Part.

Ex:- A ◇—— B

A "has-an" instance of B. B can exist without A.

Person ◇— House House exist without Person.

Desktop ◇— monitor monitor " " Desktop.

Vehicle ◇— Tyre

* **Composition:-** "has-an" or "Part of".

A strong relationship where one class (the whole) contains another class (the part), and the part can not exist without the whole.

A ◆—— B

A "has-an" instance of B. B can not exist without A.

Or, In other words, Composition implies a relationship where the child cannot exist indepent of the part.

Ex:- Body ◆— Eye Desktop ◆— motherboard

If B is destroyed then A is also destroyed. Vehicle ◆— Registration certificate

* **Inheritance:-** "is-a" relationship.

A ——▷ B A "is-a" B

* **Interface:-** A - - - - ▷ B A implements B

A - - - - ▷ B A implements B

A ——▷ B Inheritance:- A inherit from B A "is-a" B.

A - - - - B (interface, interface) Use interface: A uses interface B

A ——— B Association Bi-directional:- A and B call each other.

A ——→ Uni-directional Association: A can call B, but not vice versa

A ◇—— B Aggregation:- A "has-an" instance of B. B can exist without A.

A ◆—— B Composition:- A "has-an" instance of B. B cannot exist without A.

LLD

Date: / /
Page No.:

→ SOLID Principle.

S – Single Responsibility Principle.
O – Open/closed Principle.
L – Liskov Substitution Principle.
I – Interface Segmented Principle
D – Dependency Inversion Principle.

→ Advantages of following these Principle:-

* Avoid Duplicate code.
* Easy to Maintain.
* Easy to Understand.
* Flexible Software.
* Reduce complexity.

(2) S:- Single Responsibility Principle
"A class should have only 1 reason to change"

Example :- Below Invoice has a relationship with Marker.

Scenario:-

*1 and Suppose in future we want to change the calculateTotal() by including GST and discount also. So, Invoice class is changed due to calculateTotal(),

*2 Suppose in future we want to change the printInvoice method.

*3 suppose in future we want to save it in file.

So, all these above 3 scenarios will voilate the Single responsibility

So, To follow Single responsibility ~~to~~ while making the class we make sure that 1 class has Only 1 responsibility.

```
class Invoice {
    private Marker marker;
    private int quantity;
    public Invoice (Marker marker, int quantity){
        this.marker = marker;
        this.quantity = quantity;
    }
    public int calculateTotal(){
        int price = ((marker.price) * this.quantity);
        return price;
    }
    public void printInvoice(){
        // print invoice
    }
    public void saveTo DB{
        // Save the data into the DataBase
    }
}
```

(1) (2) (3)

3 Responsibility

Solution :=

```
class Invoice {
    private Marker marker;
    private int quantity;
    public Invoice (Marker marker, int quantity){
        this.marker = marker;
        this.quantity = quantity;
    }
    public int calculateTotal(){
        int price = ((marker.price) * this.quantity);
        return price;
    }
}
```

```
class InvoiceDao {
    Invoice invoice;
    public InvoiceDao (Invoice invoice){
        this.invoice = invoice;
    }
    public void saveTo DB{
        // save Into the Database
    }
}
```

```
class InvoicePrinter {
    prive Invoice invoice;
    public InvoicePrinter (Invoice invoice){
        this.invoice = invoice;
    }
    public void print() {
        // print the invoice
    }
}
```

Date: / /
Page No.:

(23)

O- Open / closed Principle

"Open for Extension but closed for Modification".

→ Tested
→ time in Production code.

Those class which are already tested and time "Production" should not be modified it only can be extended.

Example :- This code is not following open closed principle.

So, this is prompt to bug.

1st code Test & time in Production

```
① class InvoiceDao {
    Invoice invoice;
    public InvoiceDao { Invoice invoice ){
        this.invoice = invoice;
    }
    public void saveToDB ){
        // save into DB
    }
}
```

→ Requirement comes to save in file as well. existing code by the client.

```
→ class InvoiceDao{
    Invoice invoice;
    public InvoiceDao(Invoice invoice){
        this.invoice = invoice;
    }
    public void saveToDB ( ){
        // Save Invoice to DB
    }
    public void saveToFile ( String filename ){
        // Save Invoice in the file with the given name
    }
}
```

Solution :- Invoice DAO

- DB
- FILE
- XYZ

Infuture if any other method to save comes then

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

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

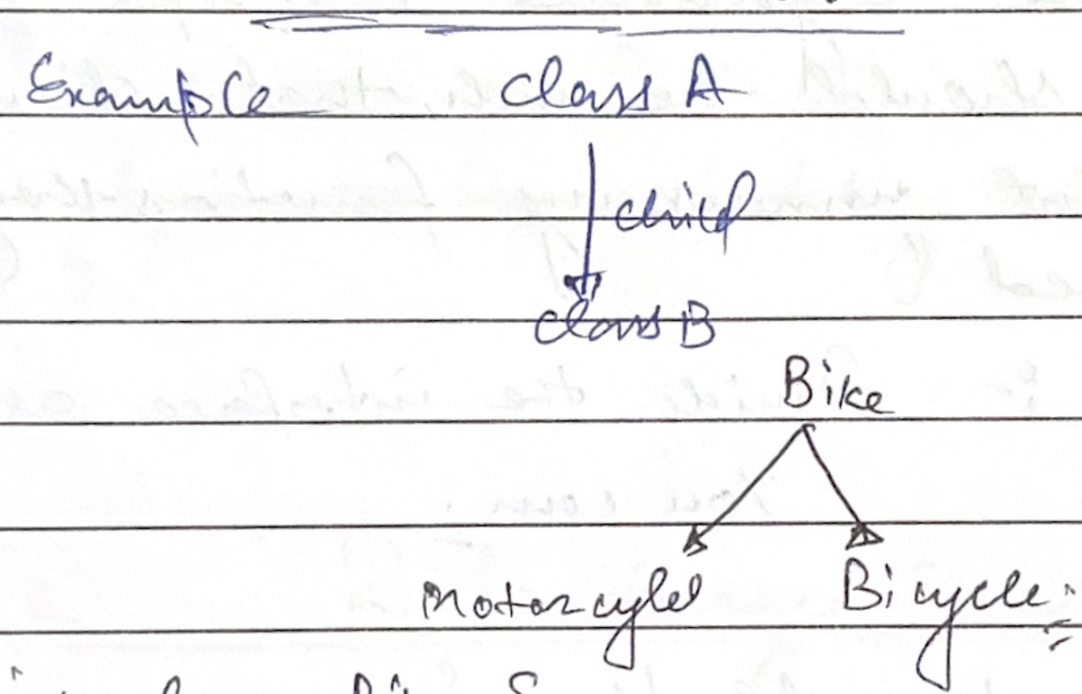
```
class FileInvoiceDao implements InvoiceDao {
    @Override
        public void save (Invoice invoice) {
            // save to file
        }
}
```

(220) L - Liskov Substitution Principle

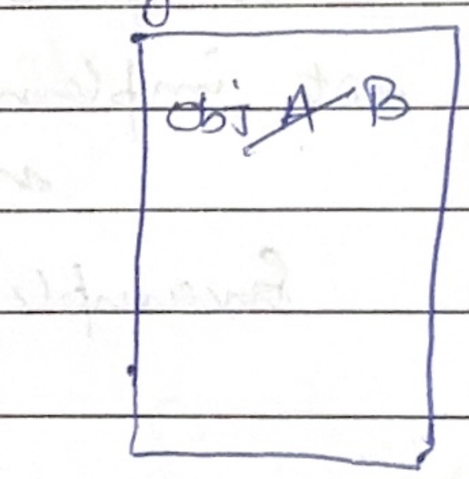
"If class B is subtype of class A, then we should be able to replace object of A with B without breaking the behaviour of the program"

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

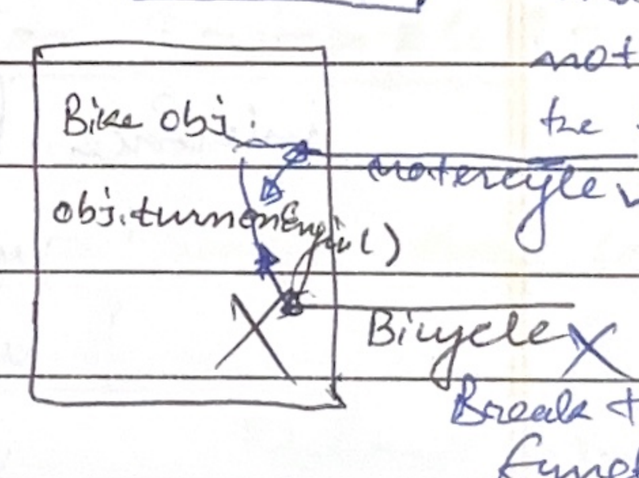
Example



A program that uses class A object



then we can replace object of class A with B and it should not break the behaviour




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

class Motorcycle implements Bike {
    boolean isEngineOn;
    int speed;

    public void turnOnEngine () {
        // turn on the Engine
        isEngineOn = true;
    }
    public void accelerate() {
        // increase the speed
        speed = speed + 10;
    }
}
```

Date: / /
Page No.:

```
class Bicycle implements Bike {
    public void turnOnEngine() {
        throw new AssertionError(detailMessage: "there is no Engine");
    }
    public void accelerate() {
        // do something
    }
}
```

Here when we replace the Bike object with Bicycle then it will narrow down the functionality & break

So, NOT follow "L" principle

(2V) I - Interface Segmented Principle

" Interface should be such, that client should not implement unnecessary functions they do not need "

Example :- Divide the interface as much as you can.

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

class Waiter implements RestaurantEmployee {
    public void washDishes() {
        // not my Job
    }
    public void serveCustomers() {
        // Yes and here is my implementation
        System.out.println("Serving the customer");
    }
    public void cookFood() {
        // not my Job
    }
}
```

Not the Job of a waiter so, it have to implement it unnecessarily.

So, Solution is divide the interface itself so that client (waiter) do not have to implement the unnecessary detail.

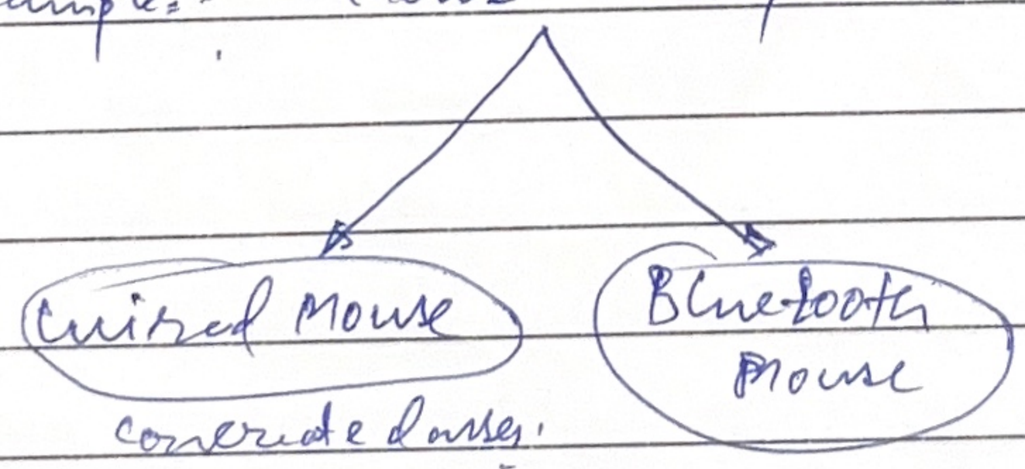
Example:

```
interface waiterInterface {
    void serveCustomers();
    void takeOrder();
}

interface chefInterface {
    void cookFood();
    void decideMenu();
}

class waiter implements WaiterInterface {
    public void serveCustomers() {
        System.out.println("Serving the customers");
    }

    public void takeOrder() {
        System.out.println("taking Order");
    }
}
```

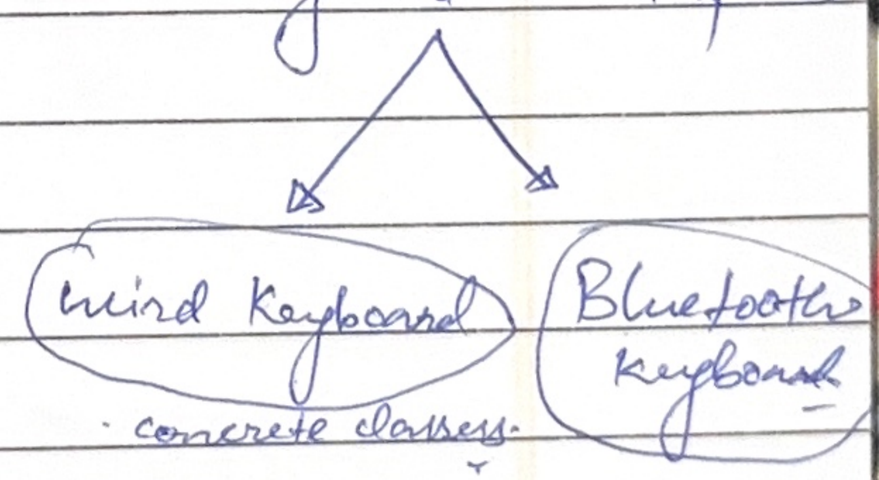
(V) D - Dependency Inversion Principle :-

"class should depend on Interfaces rather than concrete classes"

Example:- - Mouse <<Interface>>



Keyboard <<Interface>>



Date: / /
Page No.:

```
class MacBook {
    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;
    public MacBook() {
        keyboard = new WiredKeyboard();
        mouse = new WiredMouse();
    }
}
```

Assigning the concrete classes.

So, in future if we need bluetooth keyboard & mouse we cannot do it.

Bcz of create classes.

So, Not following D principle

Solution is we can replace the concreate classes with interface itself and assign the class with constructor injection.

```
class MacBook {
    private final Keyboard keyboard;
    private final Mouse mouse;
    public MacBook(Keyboard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }
}
```

Interfaces

So, in future we can pass wired or bluetooth not hardcoded

Date: / /
Page No.:

Lec-2:

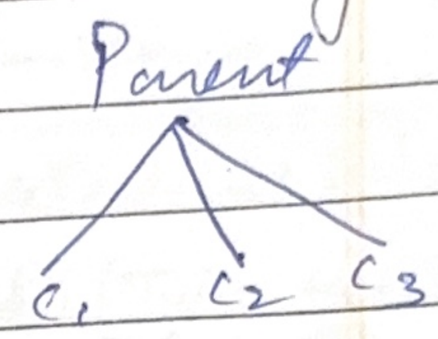
Solution of Liskov Substitution Principle:

"If class B is subtype of class A, then we should be able to replace object of A with B without breaking the behaviour of the program".

Parent parent obj = new C1();

new C2();

then it should be replaceable.



Design Patterns.

Date : / /
Page No. :

Design Patterns are typical solutions to commonly occuring problems in software design. They are like premade blueprints that you can customize to solve a recurring design problem in your code.

With the help of Design Pattern we can write software:
- → Managable
- → Scaleable
- → Reusable

Types of Design Patterns.

(1) Behavioural Design Patterns.

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Types :-

(i) Strategy Design Pattern
(ii) Observer Design Pattern
(iii) State Design Pattern using Vending Machine Example.
(iv) Chain of Responsibilities Design Pattern with Logging Example.
(v) Command DP with Undo/Redo functionality Example.
(vi) Template mtd DP
(vii) Momento DP
(viii) Interpretor DP
(ix) Mediator DP with Online Auction Example.
(x) Null object DP
(xi) Iterator DP

* Behavioural Design Patterns : These design Patterns are specifically concerned with communication between objects.

Date: / /
Page No.:

(2) Creational Design Patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code. These DP provide ways to create objects while hiding the creation logic, instead of instiating objects directly using new operator. This gives the program more flexibility in deciding which objects need to be created for a given use case.

(i) Factory method
(ii) Abstract factory DP
(iii) Builder DP
(iv) Prototype DP
(v) Singleton DP

(3) Structural Design Patterns.

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

(i) Adapter DP
(ii) Bridge DP
(iii) Composite DP
(iv) Decorator DP
(v) Facade DP
(vi) Flyweight DP
(vii) Proxy DP

These design Patterns deals with class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionality.

Date : / /
Page No. :

Behavioural DP.

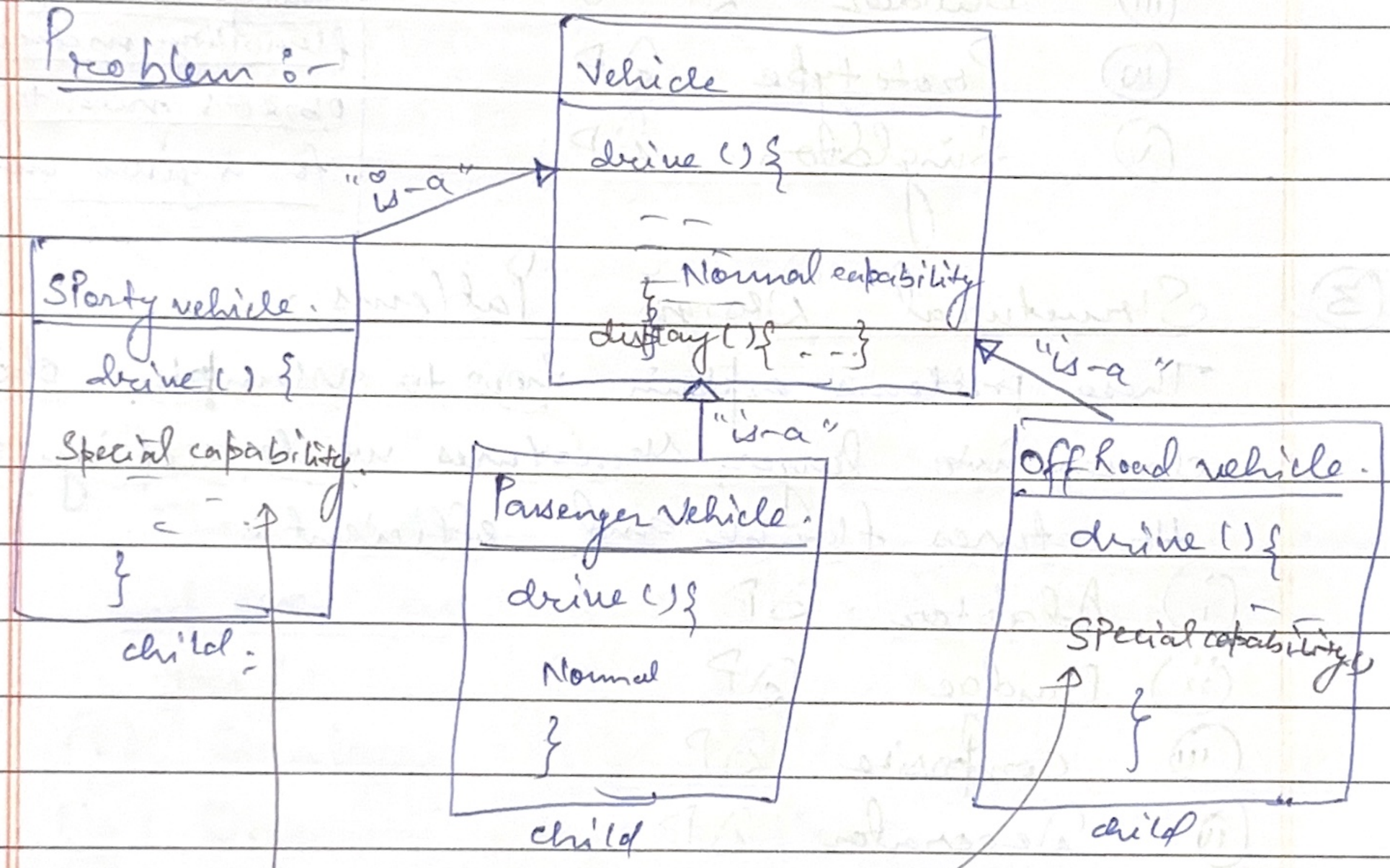
(a 2)

Strategy Design Pattern

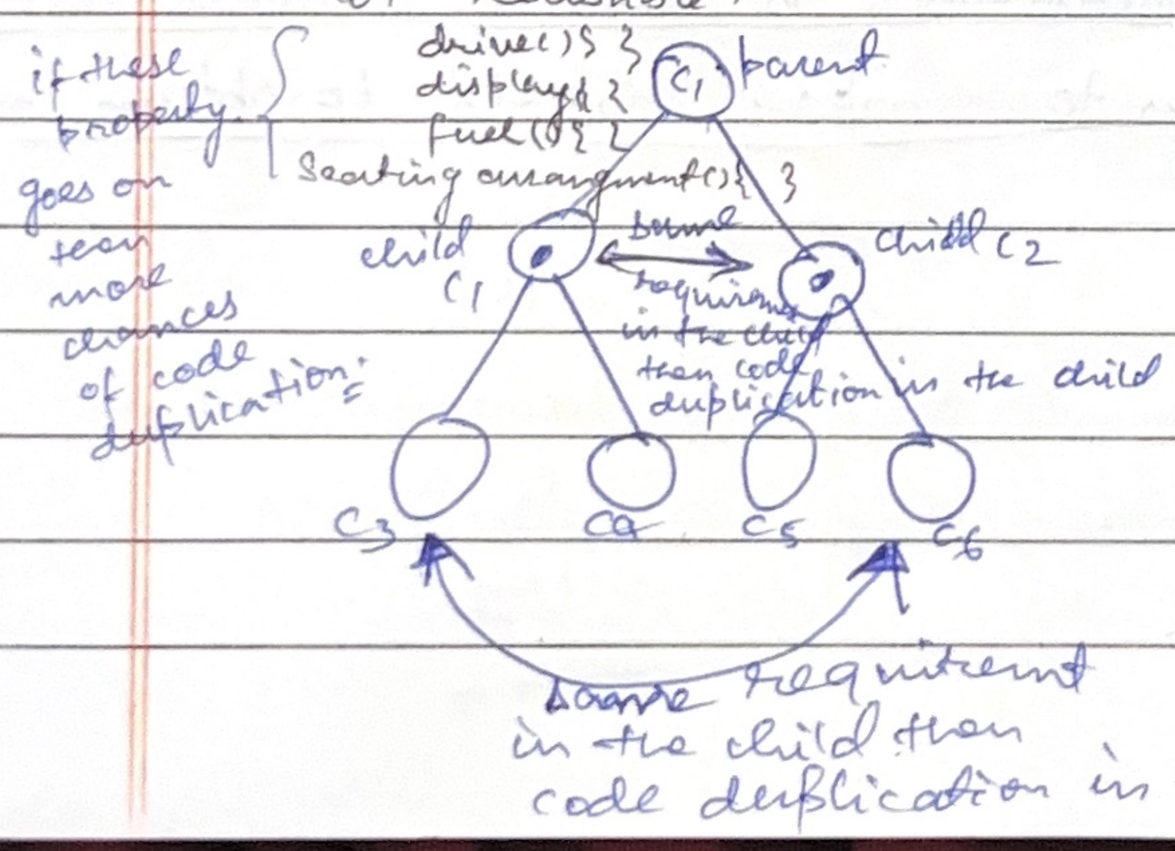
"is-a" → inheritance
→ "has-an"

Strategy is a behavioural design pattern that lets you define a family of algorithms, puts each of them into a separate class or interface, and make their objects interchangeable.

Problem :-



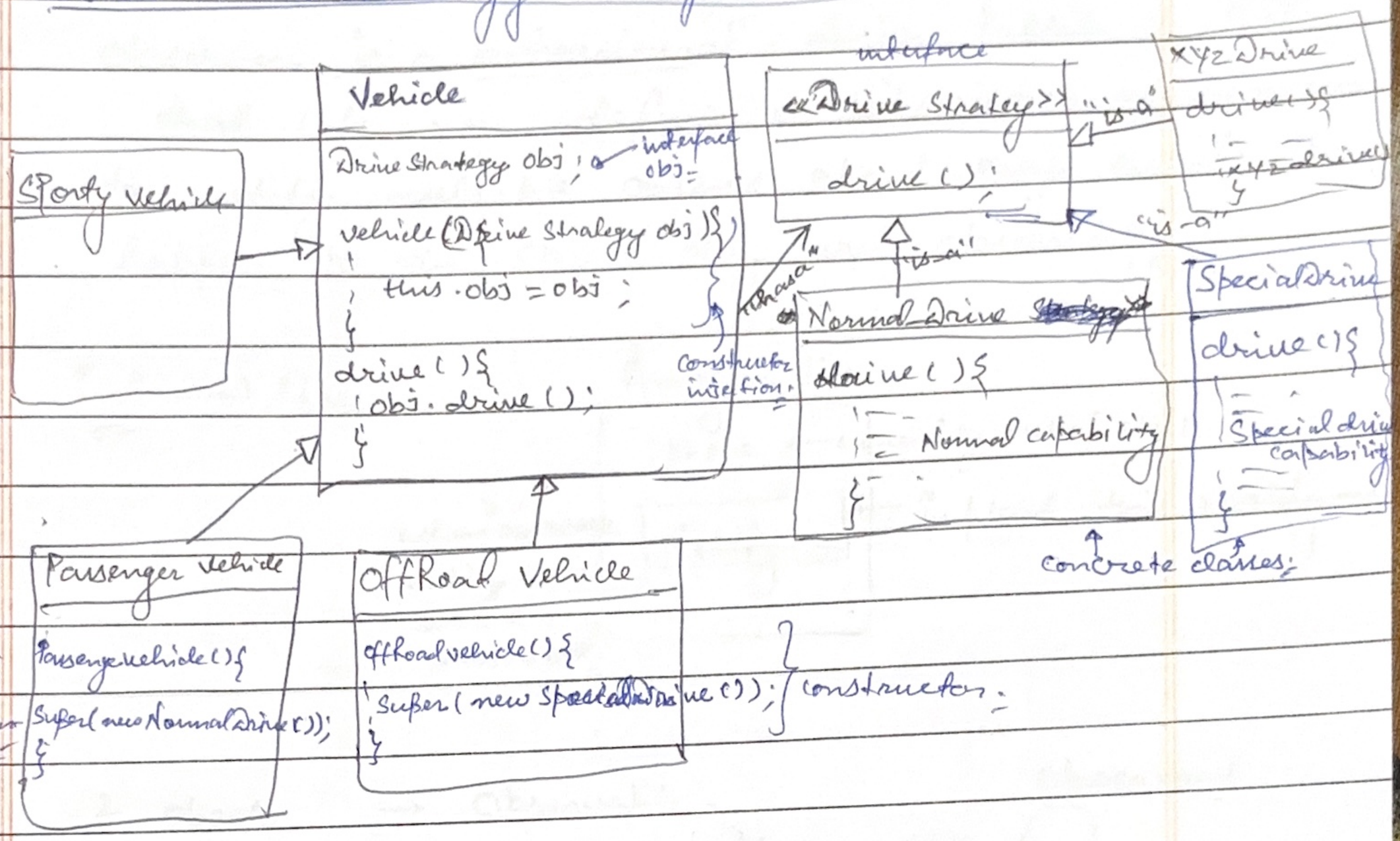
If there would a situation where two or more child having the same requirement like here they both need special capability then in that case there would be code duplication in the child so, code is not reusable.



At the same level when child is not using the parent property they are using their own property and if they are same then there is an issue of code reusability & scalibility.

Date : / /
Page No. :

Solution is Strategy Design Pattern.



Date :
Page No. :

(22)

Observer Design Pattern.

(Walmart interview Qns = Behavioral Design Pattern)

Observer is a behavioral design pattern that lets you define a subscription mechanisms to notify multiple objects about any events that happen to the object they are observing.

→ interview Qns.

Amazon Site.

Iphone — currently out of stock

Notify me — implement this Notify me.

when it is available notify to all ~~who has~~ subscribed.

→

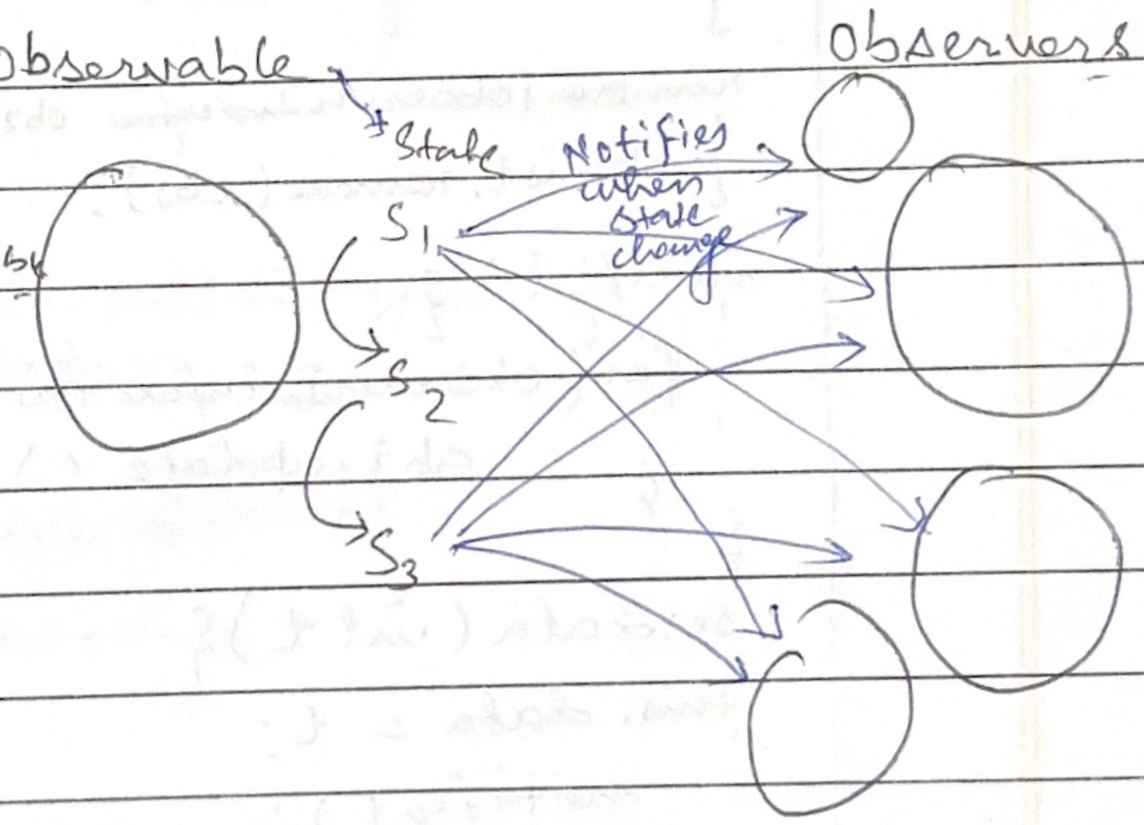
2 objects. → Observable → Observers

State change of observable

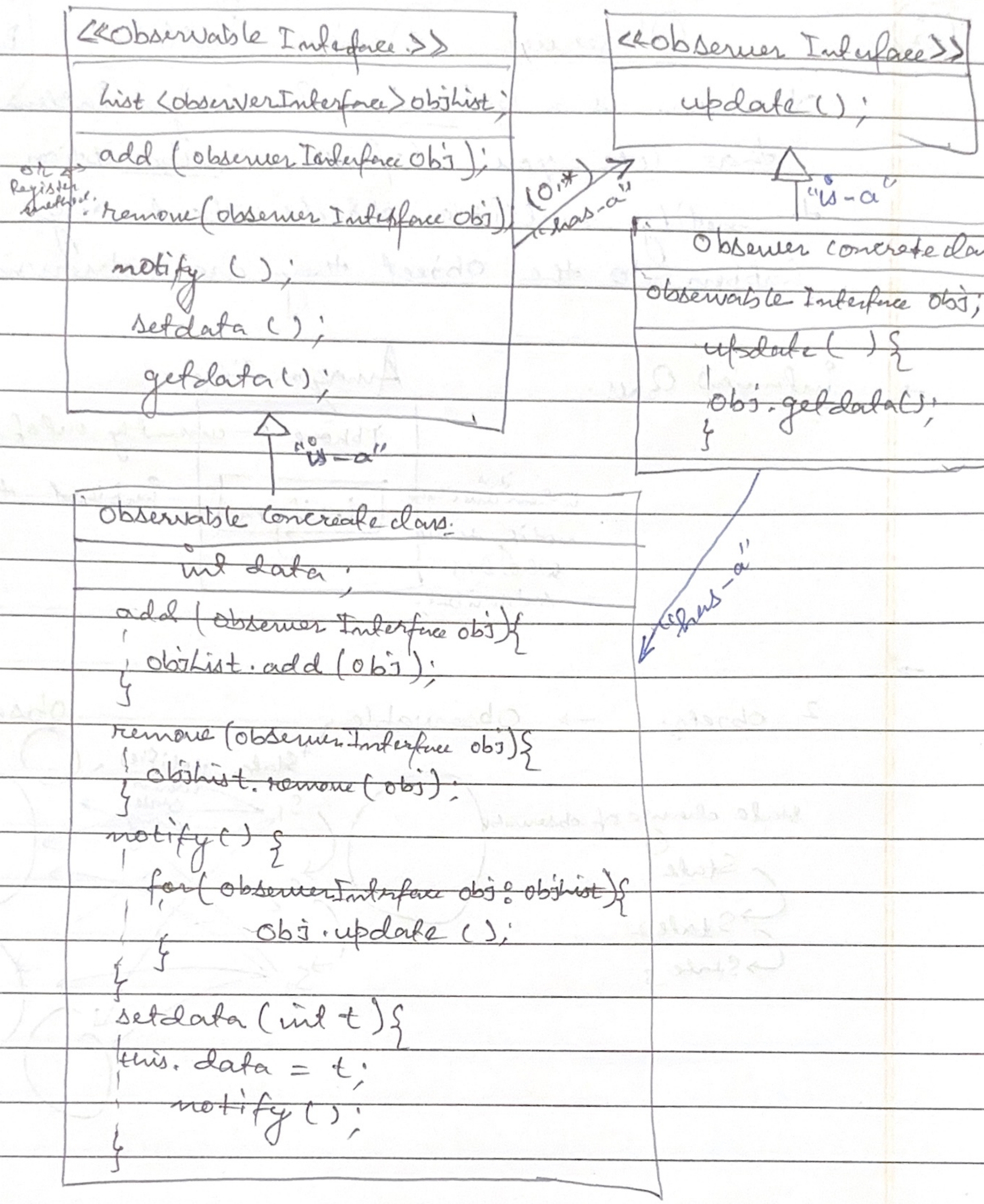
State 1

State 2

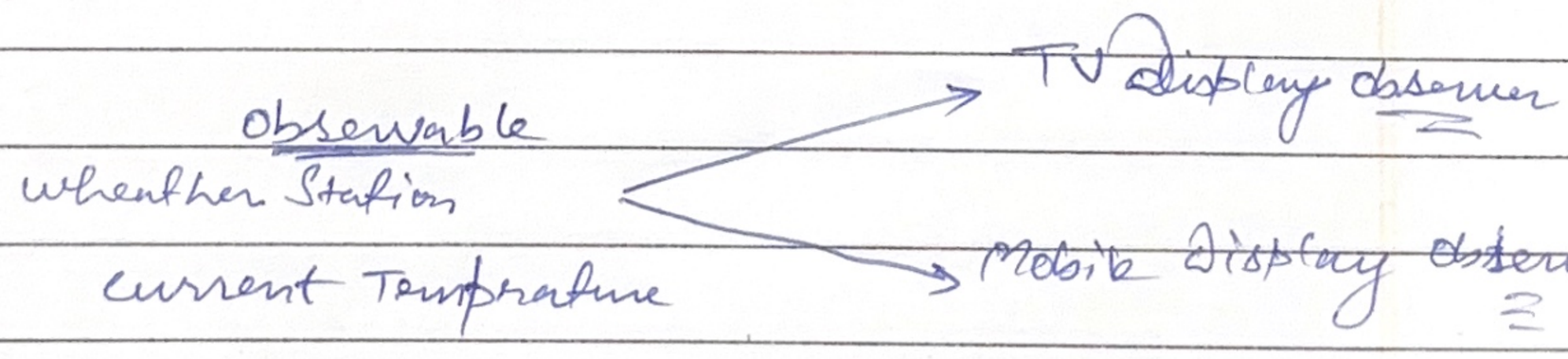
State 3



Date: / /
Page No.:



Example :- Weather Station.



Date : / /
Page No. :

weatherStation

```
<< WS observable >>
add (DisplayObserver obj);
remove (DisplayObserver obj);
notify ();
setTemp();
```

"has-a"

```
<<DisplayObserver>>
update();
```

```
TVDisplay obs
```

```
mobileDisplayObservable:
WSobservable1 obj;
mobileDisplayObservable(WS observable0){
  this.obj = 0;
}
update (){
  obj.getdata();
}
```

"has-a"

```
class
WS observable1 implements WS observable
List<DisplayObserver> displayhist;
int temp;
add (){
displayhist
}
remove{

}
notify(){
for (Display observer obj: displayhist){
  obj. update();
}
setdata( int newdata){
this. temp = newdata;
notify();
}
```

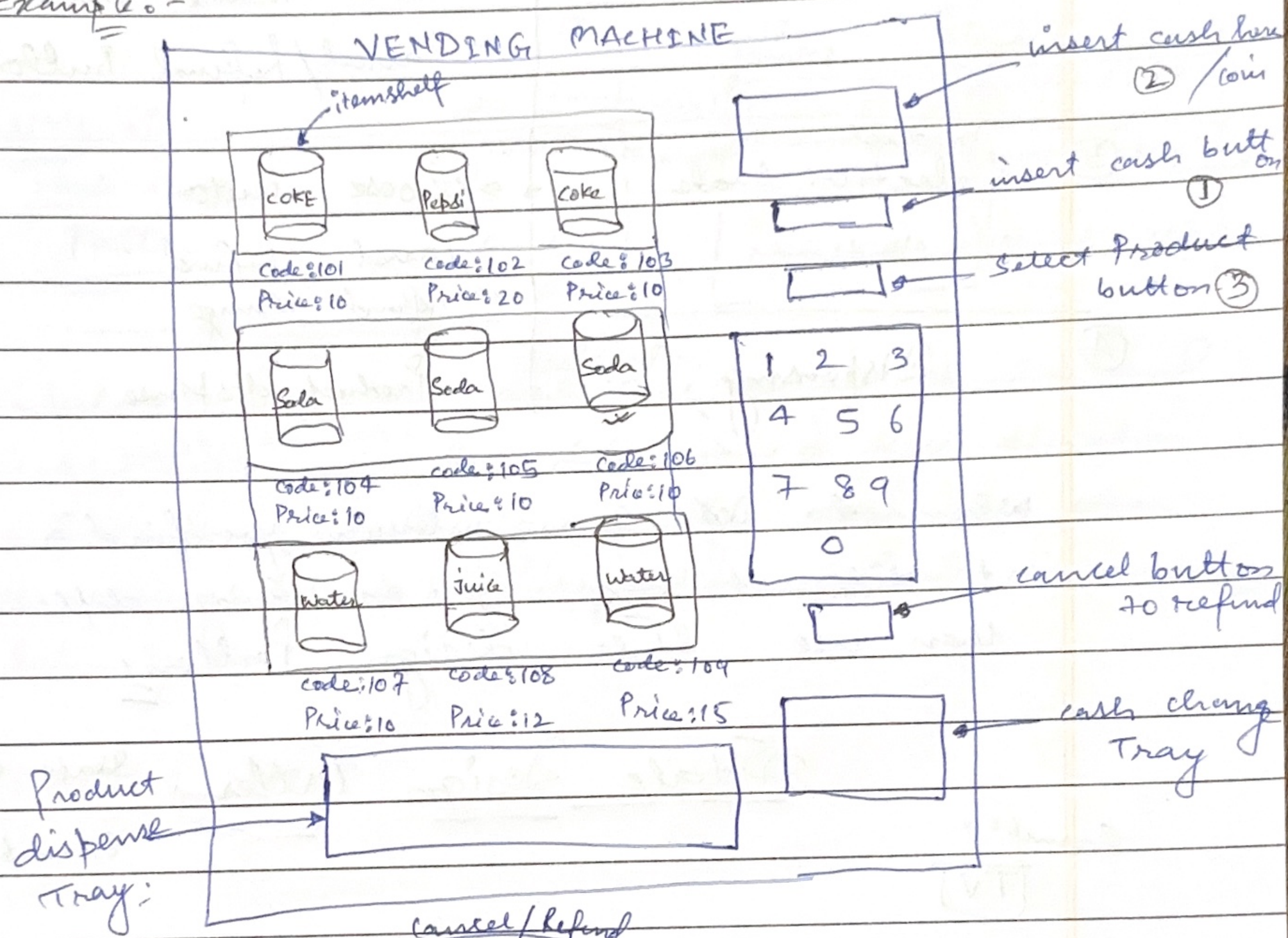
(222) State Design Pattern

Date: / /
Page No.:

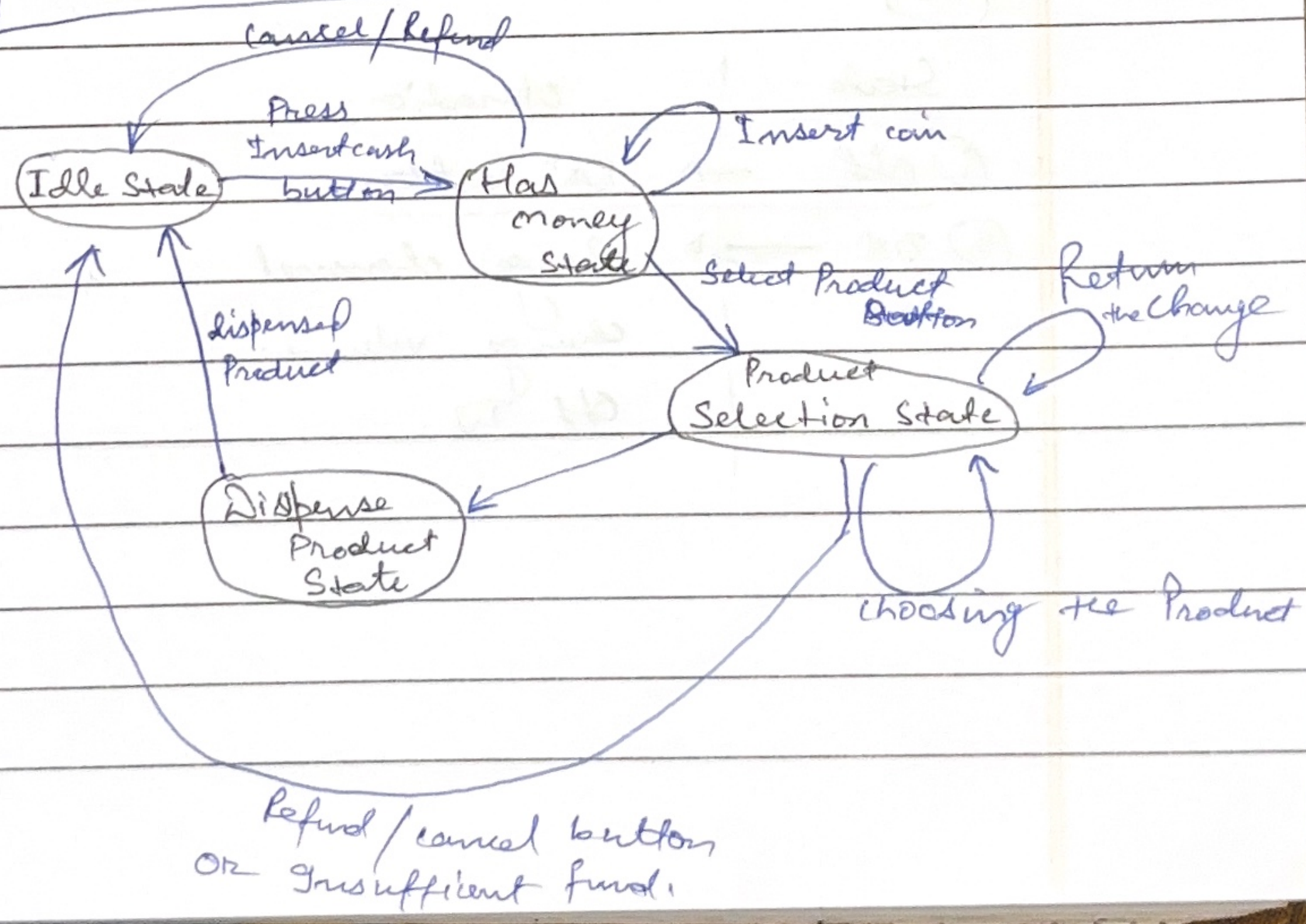
→ State is a behavioural design Pattern that lets an object alter its behaviour when its internal state changes. It appears as if the object changed its class.

→ The State Design Pattern is closely related to the concept of a Finite-State machine.

Example:-



Happy Flow:-



Date: / /
Page No.:

** Vending Machine are doing State wise → Operation

Vending machine:

State	Operation
① Idle	Press Insert cash Button
② Has money →	→ Insert coin, → Select Product Button → Cancel / Refund Button
③ Product Selection State	→ choose Button → cancel / Refund → refund change
④ Dispensing:	→ Product dispense

When at diff In an interview you find a pattern like at different state you are doing different operations then use State design Pattern.

State Design Pattern

State wise operations.

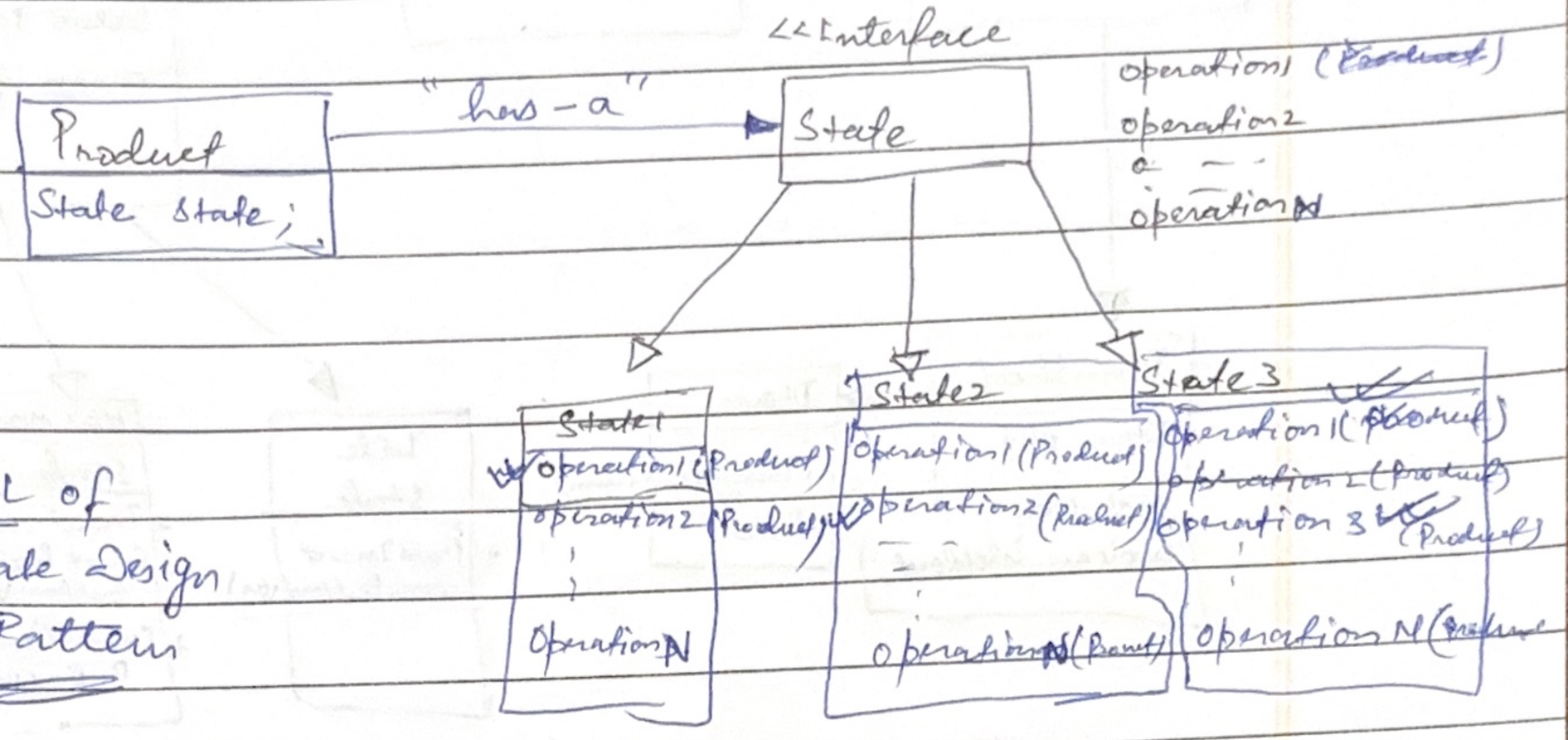
Example:

Example:

TV

State	Operation:
(i) off →	ON button
(ii) ON →	change channel change volume. Off TV

Suppose a Product has different state then and a State has different operations:



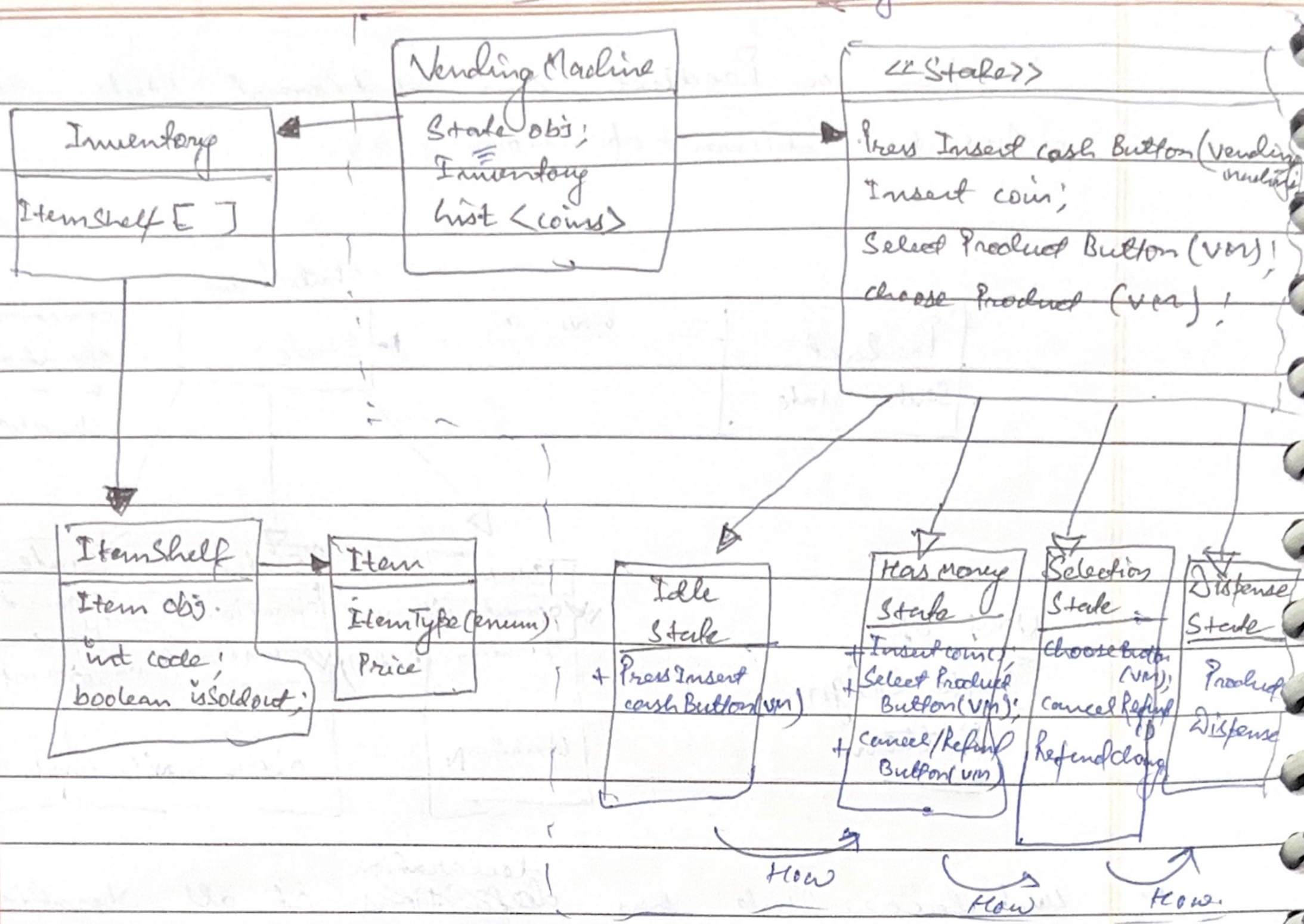
UML of State Design Pattern

→ <interface>> State has ~~definitions~~ declaration of all operation and State1, State2, State3 etc: will only has definitions of those operation which it is performing correctly. and all other operations of State1, State2, ... State3 will either default or exception behaviour;

UML of VM.

Date: / /
Page No.:

State design Pattern



UML for Vending Machine.

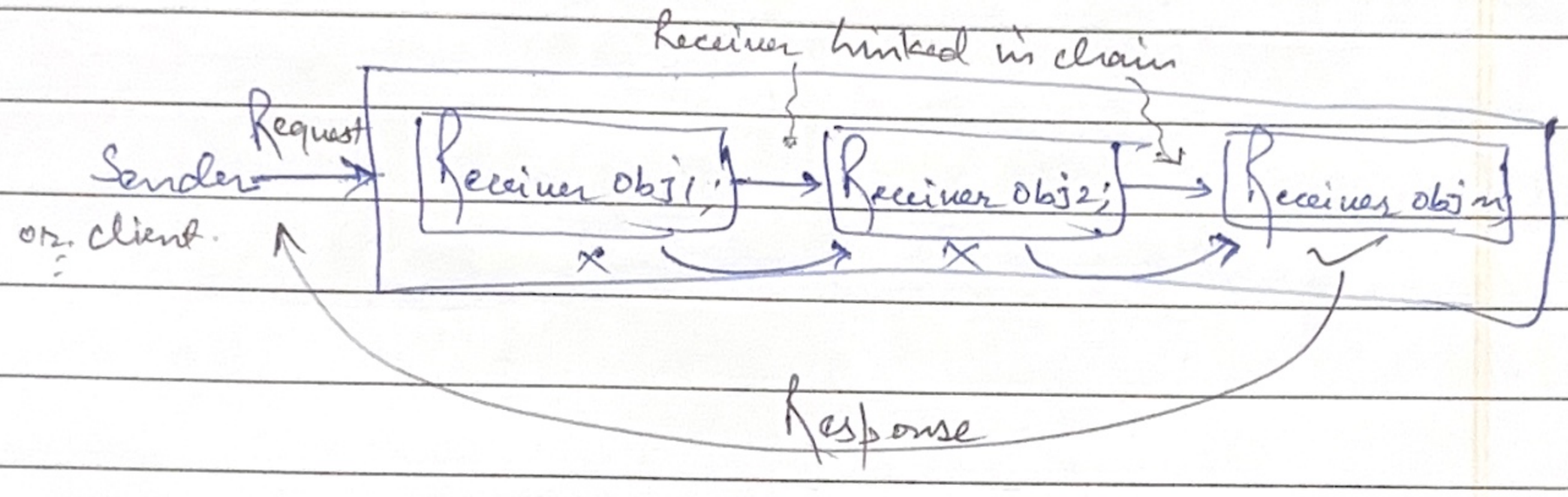
(iv) Chain of Responsibility :-

Date : / /
Page No. :

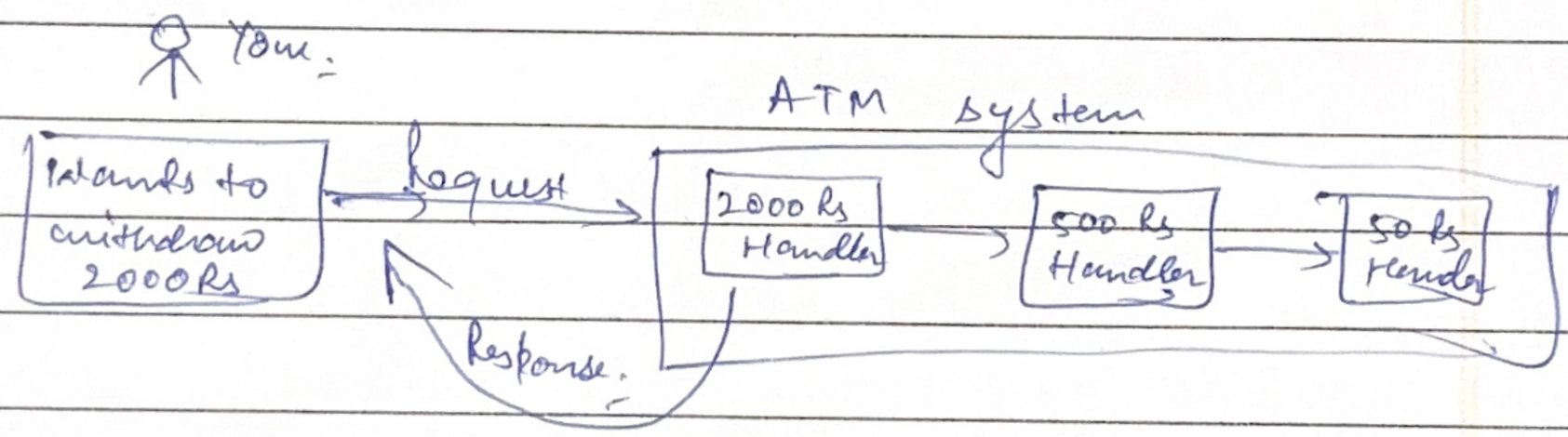
Chain of Responsibility is a behavioural design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Application Usage :-

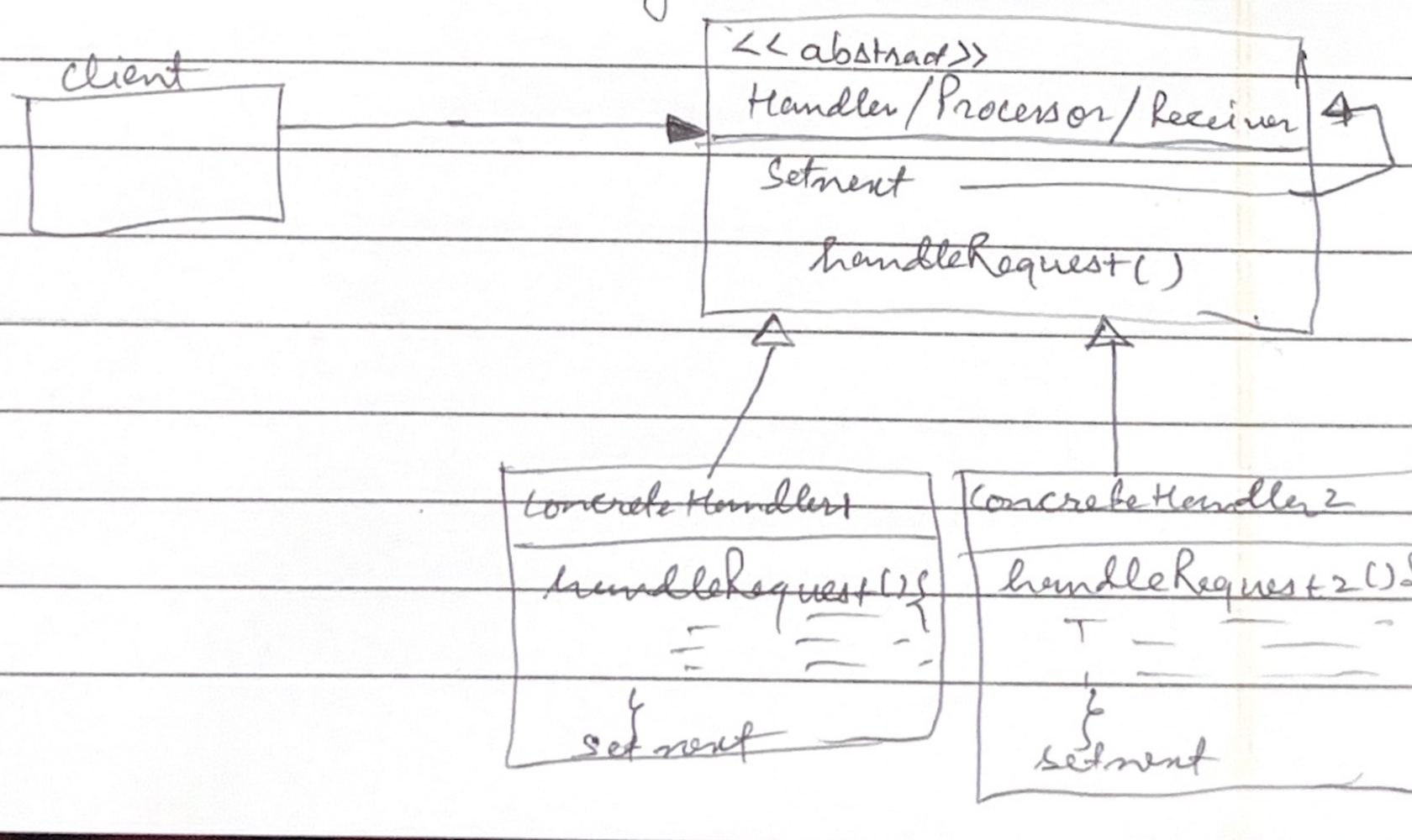
→ ATM / vending machine.
→ Design Logger (Amazon intern Q's). Logging.



→



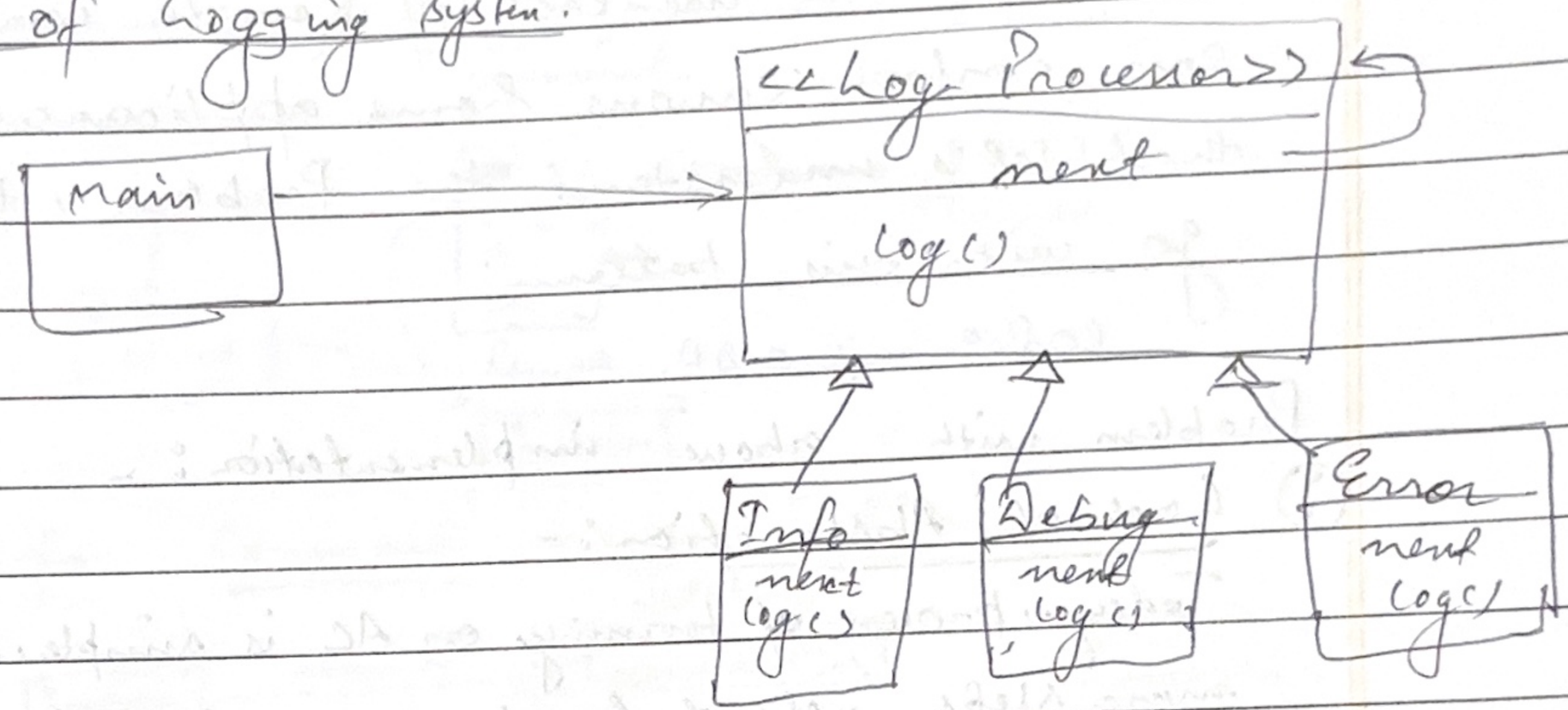
UML of chain of Responsibility.

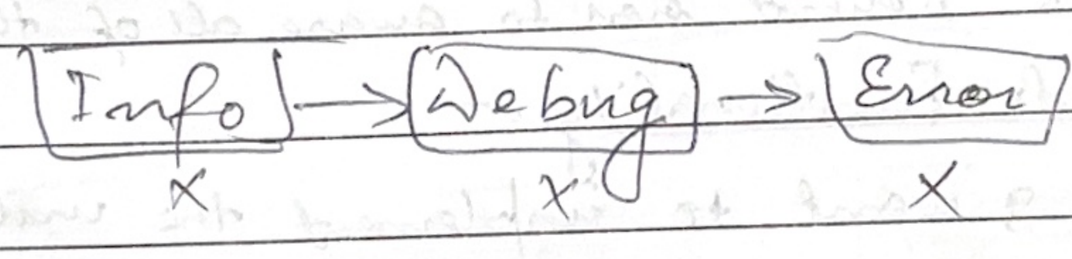


Example: Design Logging System.

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

UML of Logging system.





(V) Command Design Pattern

Date: / /

Design Undo, Redo, feature

Command is a behavioural design Pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queues a request's execution, and support undoable operations.

Let's take the use-case of Remote control which can control various home appliances and with that let's understand the Problem, then we will go with this pattern.

Code:- with CP.

Problem with above implementation:-

(i) Lack of Abstraction:-

Todays, process of turning on AC is simple, but if there are more steps, client has to aware all of that, which is not good.

(ii) Undo / Redo Functionality:-

what if I want to implement the undo/redo capability. How it will be handled.

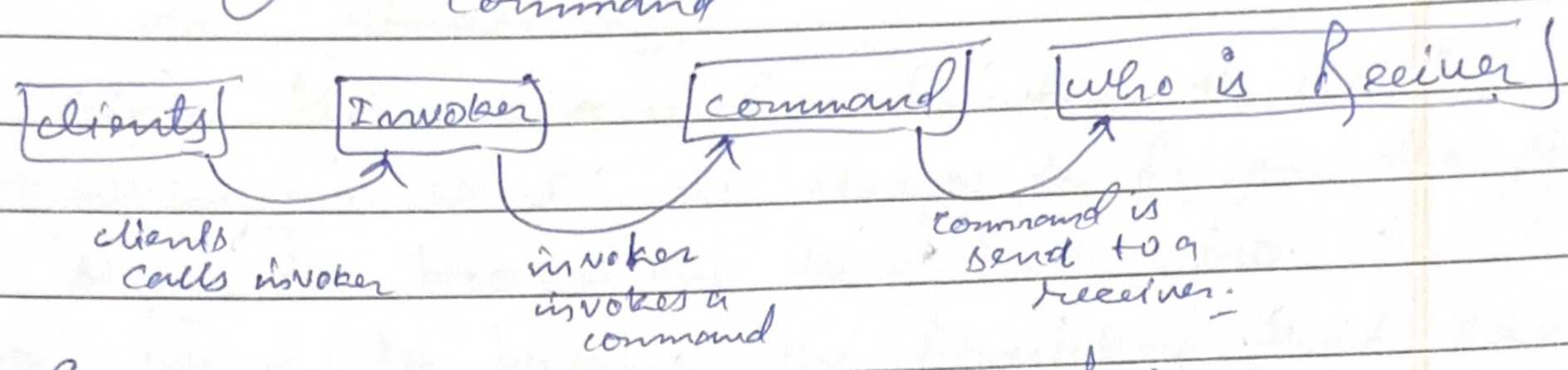
(iii) Difficulty in code maintainance:-

what if in future, we have to support more commands for more devices example bulb. ma

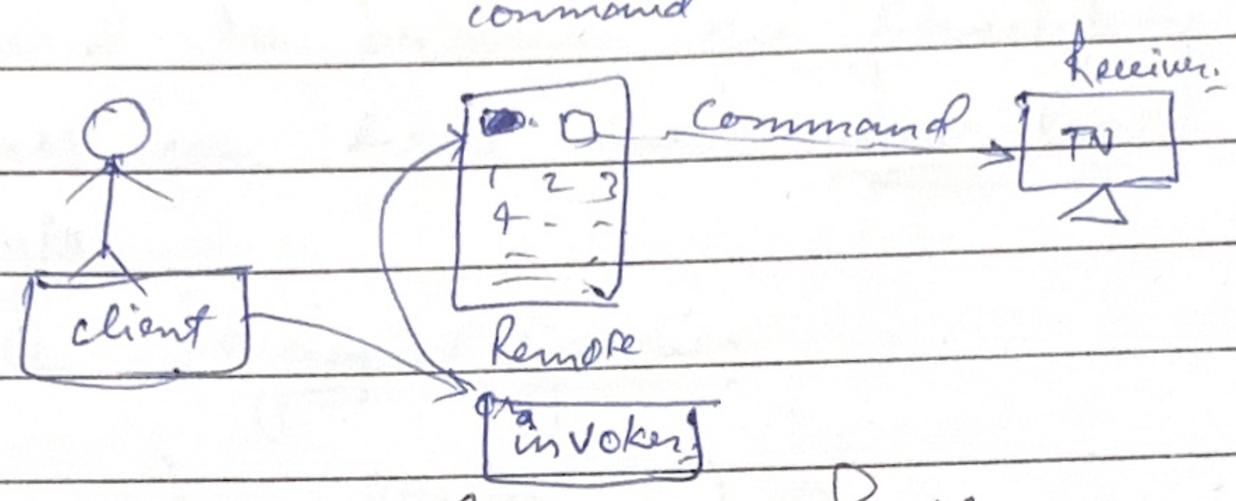
→ How command design Pattern solves it?

It separates the logic of :

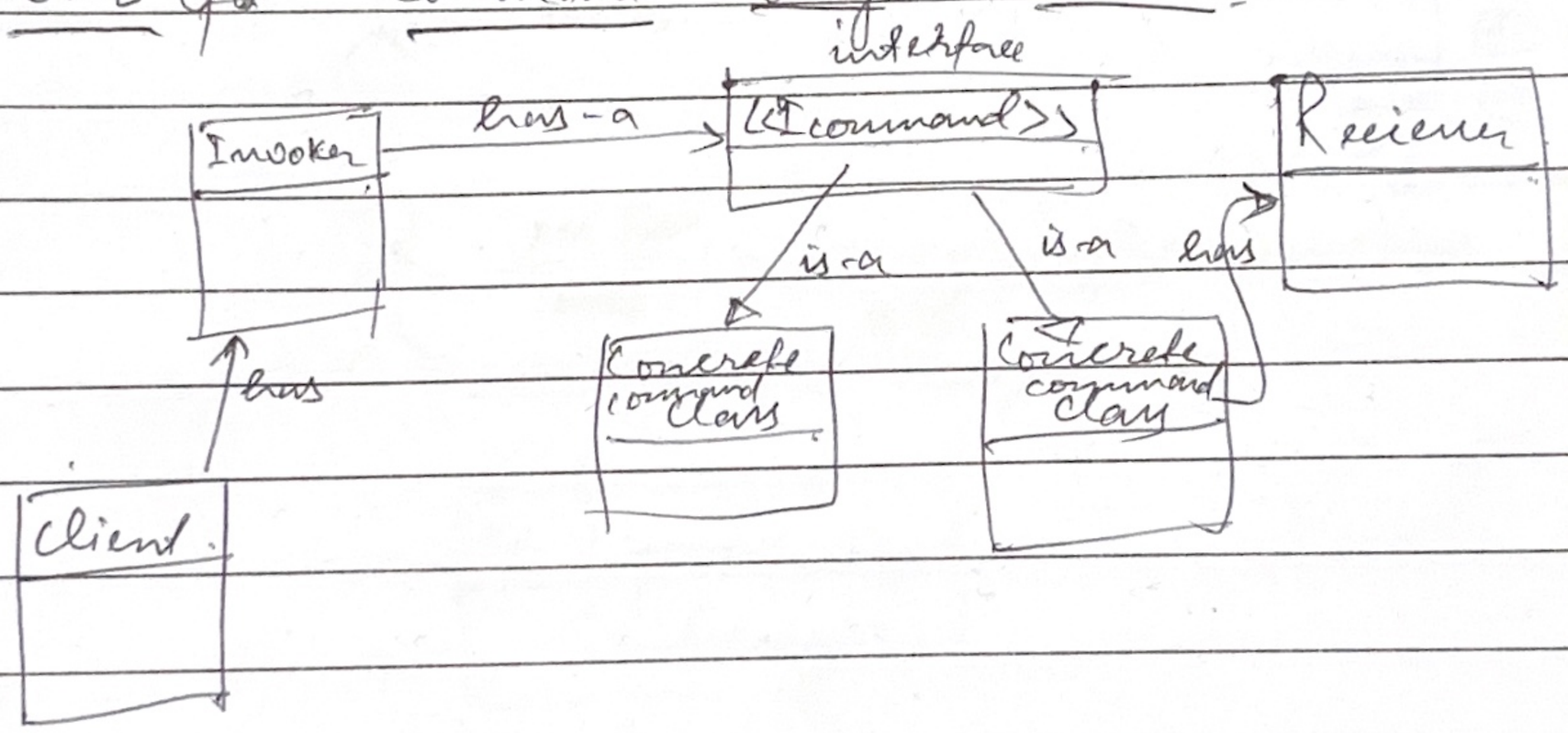
- (a) Receiver
- (b) Invoker
- (c) Command



Ex:



UML of a Command Design Pattern:



(vi) Template Design Pattern

Template Design method is a behavioural design pattern that defines the skelton of an algorithm in the superclass but lets subclass overrides specific steps of algorithm without changing its structure.

* Why it's required and when to use?

→ When you want all classes to follow the specific steps to process the task but also

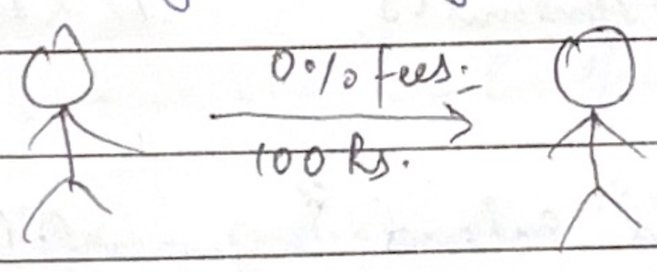
→ Need to provide the flexibility that each class can have their own logic in that specific steps.

Example Payment system:

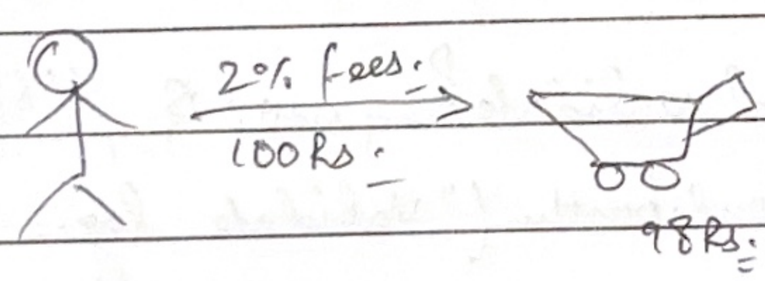
Payment class (Send Money) follows these steps:

(i) validate Req.
(ii) Debit money
(iii) calculate fees
(iv) Credit money

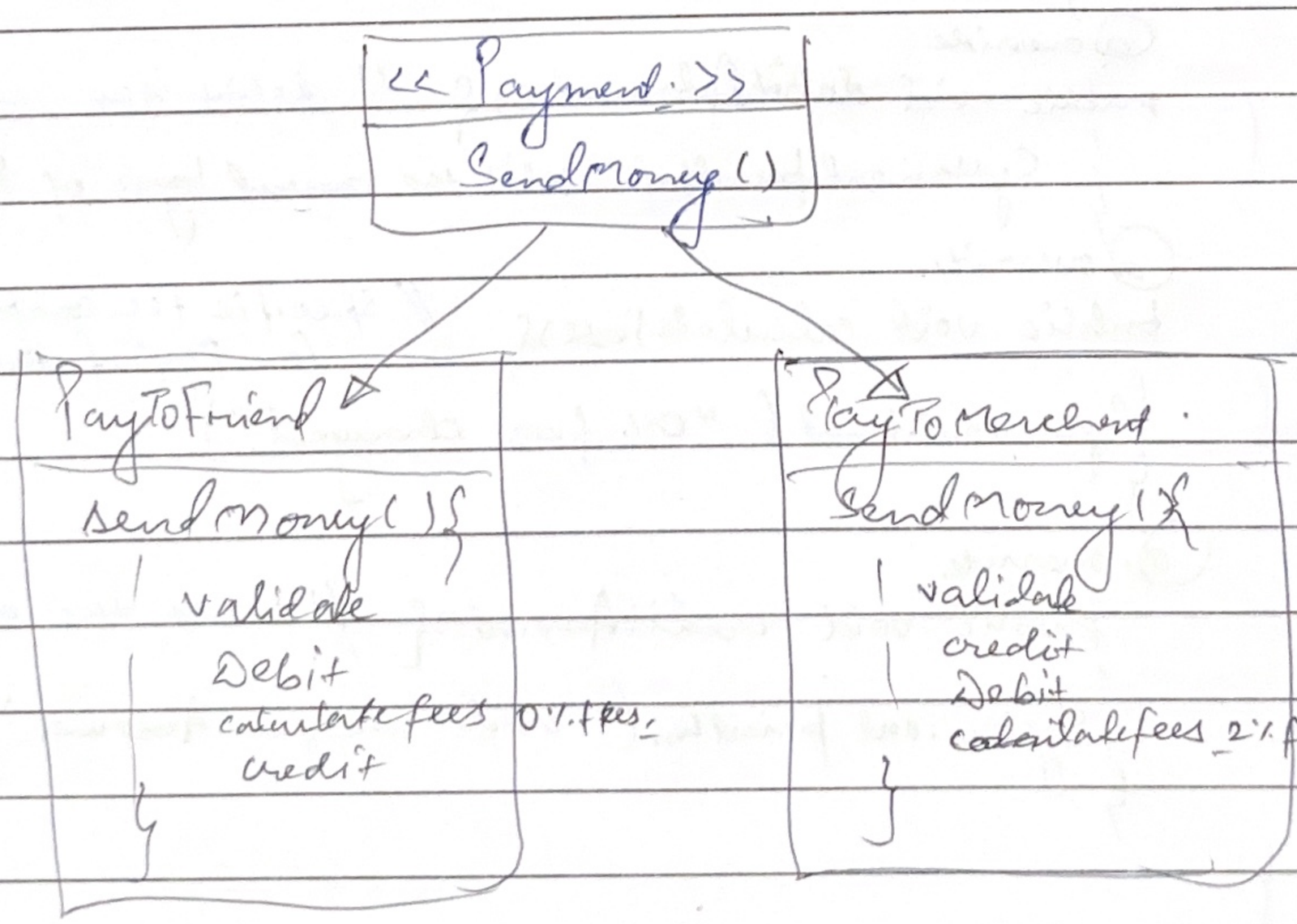
in an order.



PayToFriend flow



PayToMerchant Flow



Date: / /
Page No.:

in main method or client:

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

Payment Flow is an abstract class:

```
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(){
        validateRequest();      // step1
        debitAmount();          // step2
        calculatFees();         // step3
        creditAmount();         // step4
    }
```

No child class can override it:

```
public class PayToFriend extend 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.print("0% fees charged");
    }
    @override
    public void creditAmount(){ // credit the amount logic
        System.out.println("credit the full Amount");
    }
}
```

Date: / /
Page No.:

```
public class PayToMerchant extend PaymentFlow {
    @Override
    public void validateRequest() {    // Specific validation for
                                          MerchantPayment
    System.out.println("validate logic for PayTo Merchant");
    }

    @Override
    public void debitAmount() {    // debit the amount.
    System.out.println("Debit the Amount logic
    }
    @Override
    public void calculatfees() {    // specific fee computation logic
                                       for merchant.
    System.out.println("2% fees charged");
    }

    @Override
    public void creditAmount() {    // credit the amount logic
    System.out.println("credit the remaining Amount");
    }
}
```

(vii) Momento Design Pattern

Date: / /
Page No.:

also known as Snapshot design pat.

→ Momento means Snapshot. D.P.

→ Momento is a behavioural design Pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Why it's required and when to use:-

→ Provide an ability to revert ~~the~~ an object to a previous state i.e ~~and~~ UNDO capability.

→ And, it does not expose the object's internal implementation.

Major Components in "Momento Pattern"

ⓐ Originator ⓑ Momento ⓒ Caretaker.

→ ⓐ Originator :-

→ It represents the object, for which state need to be saved and restored.

→ Expose methods to save and restore its state using Momento object.

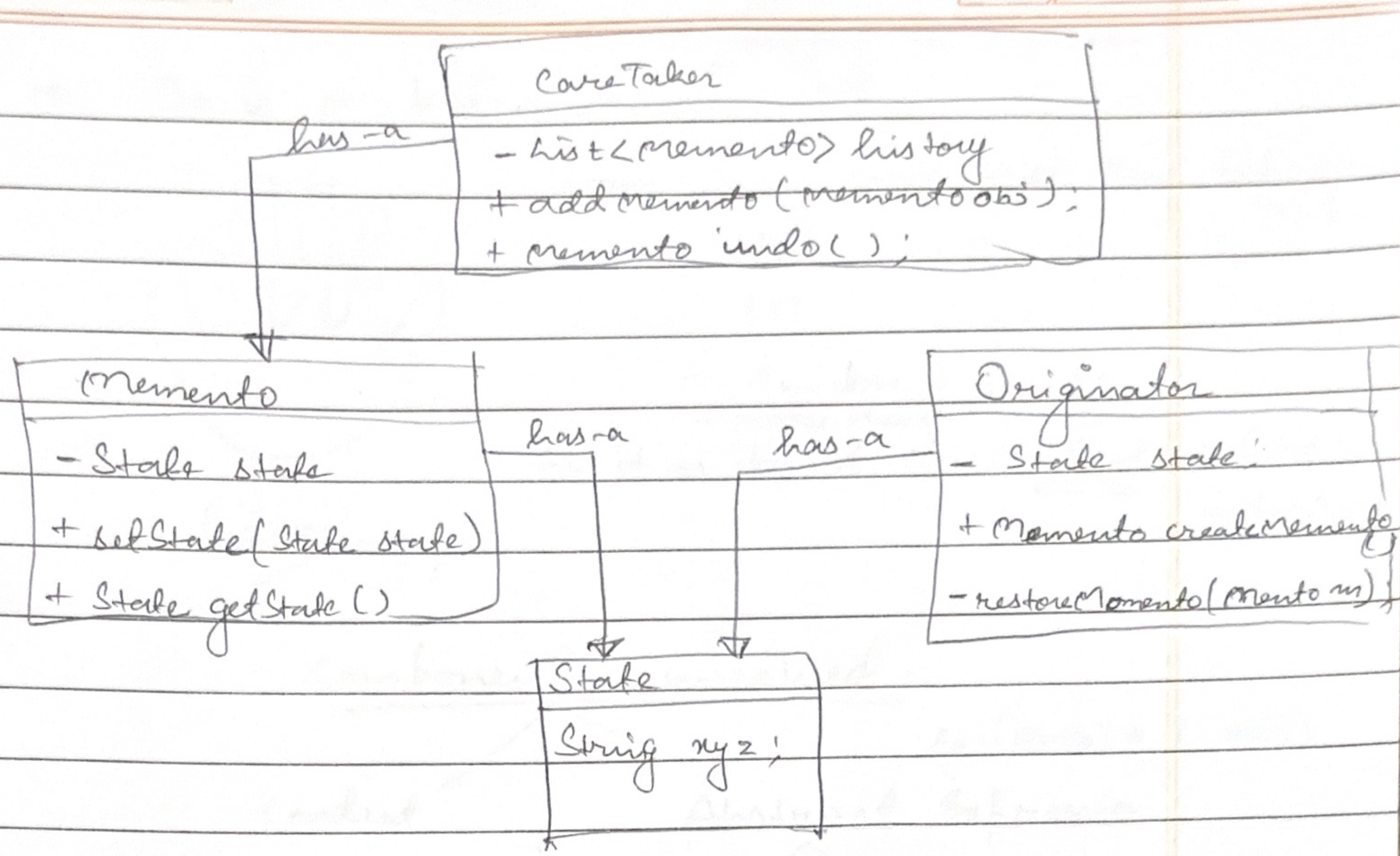
→ ⓑ Momento :-

→ It represents an objects which holds the state of the originator.

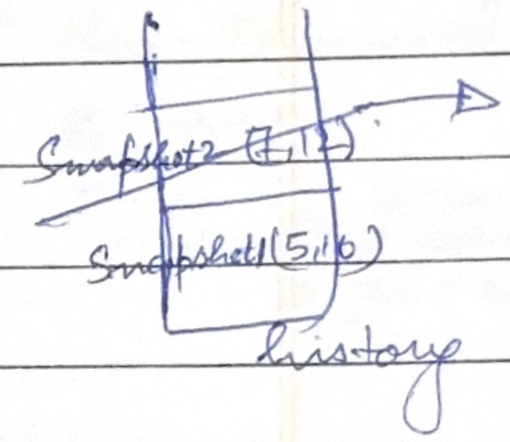
→ ⓒ CareTaker :-

→ Manages the list of state (i.e. list of momento)

** Originator do not expose its implementation its know only how to save its state.



UML of Memento DP.



(viii) Intrepreter DP.

Date: / /
Page No.:

→ It is a behavioural DP.

to understanding the expression.

Some can interpret this expression as:

- stop
- Hi
- Number 5
- raise Hand.

etc.

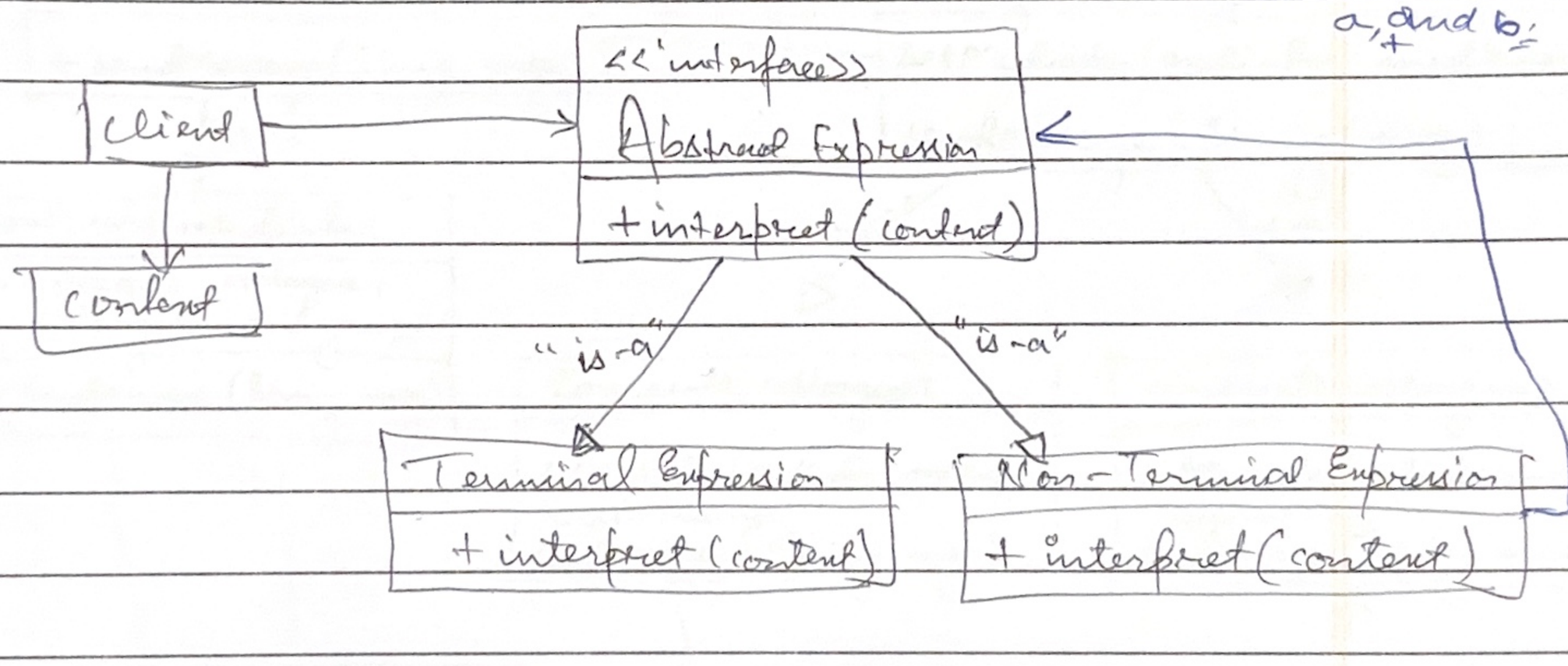
So, it is based on content what we interpret.

* Components involved.

- Content
- Abstract Expression

Eg. ((a+b) * (c*d))

  - Terminal Expression
    Eg. a, b
  - Non-Terminal Expr.
    Eg. a+b
    further it can be divided into a, + and b.



UML of Interpreter DP

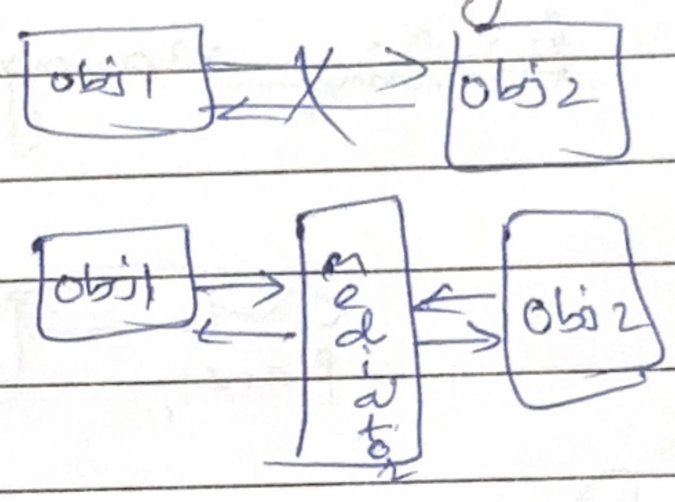
Eg: (a*b) Expression.

(ix) Mediator DP with Online Auction Example

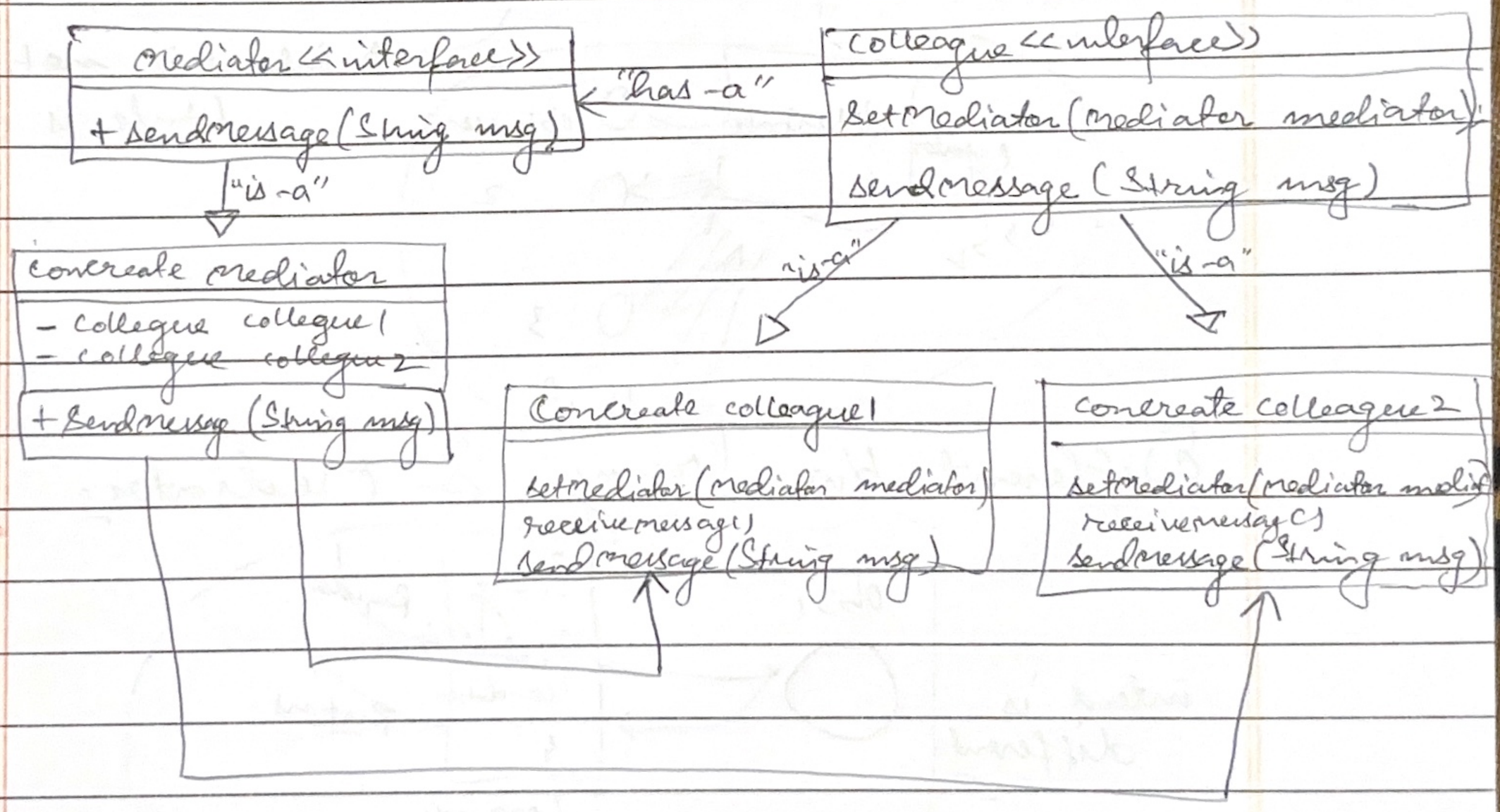
Also known as Intermediary, Controller.

→ Mediator is a Behavioural DP that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

→ It encourage loose coupling by keeping objects from referring to each other explicitly and allows them to communicate through a mediator object.



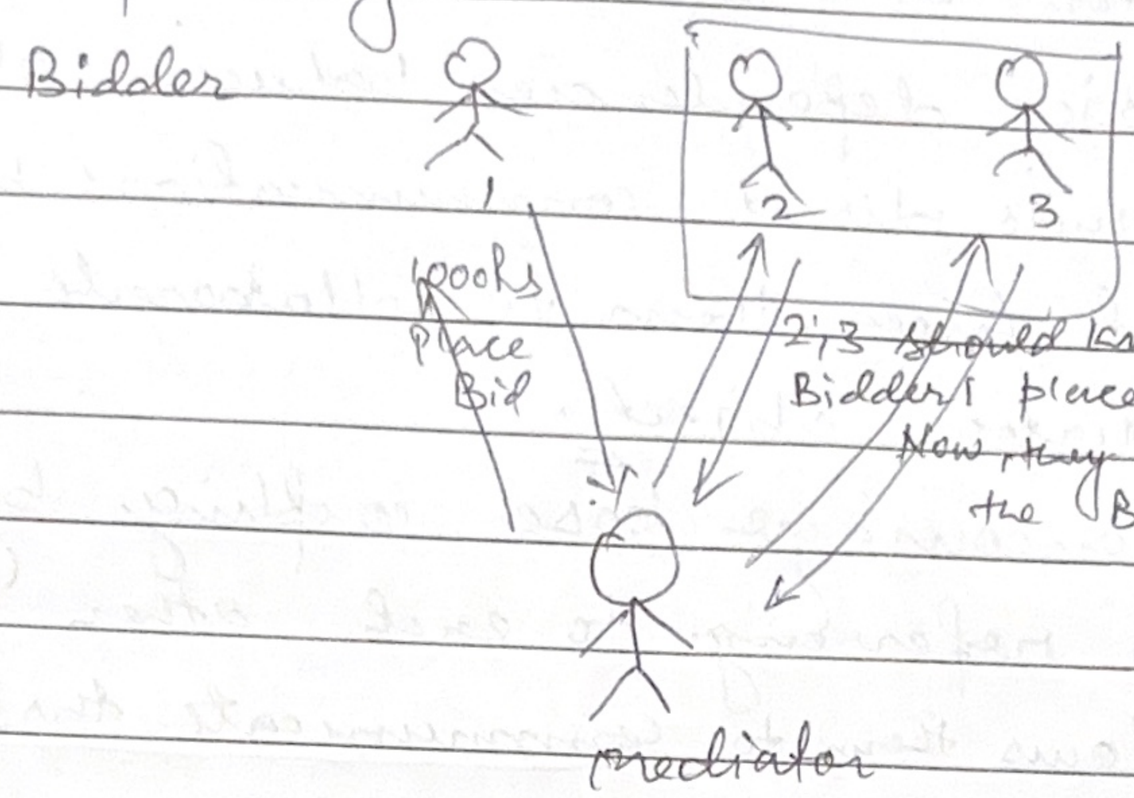
UML



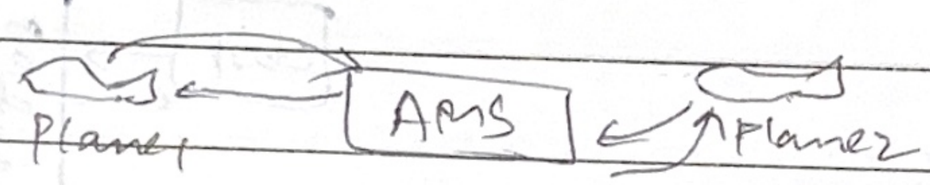
Date : / /
Page No. :

Example:

Online Auction System



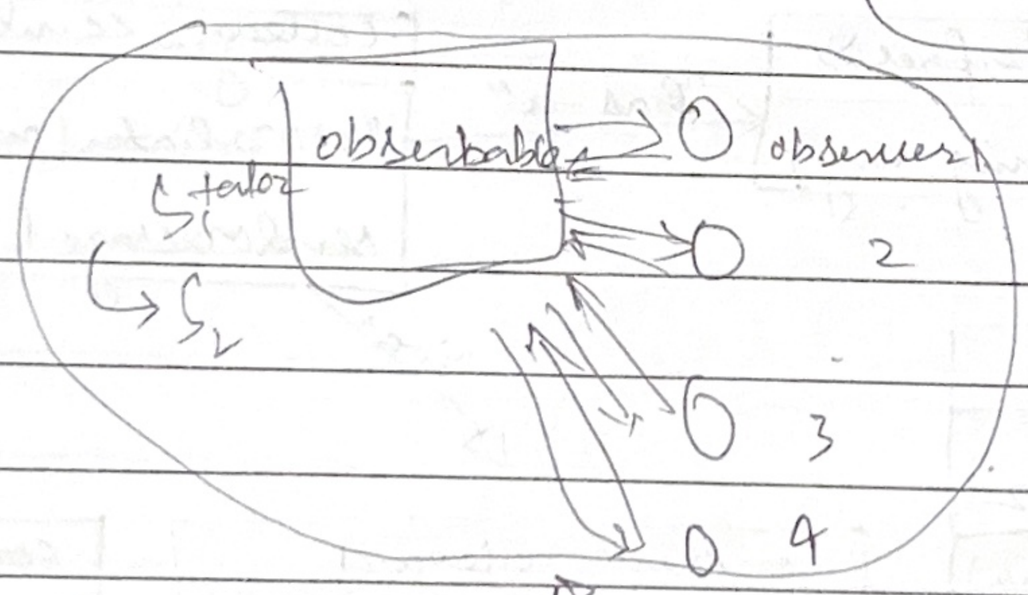
Airline Management System.



No, two plane talks to each other, but only talk to AMS.

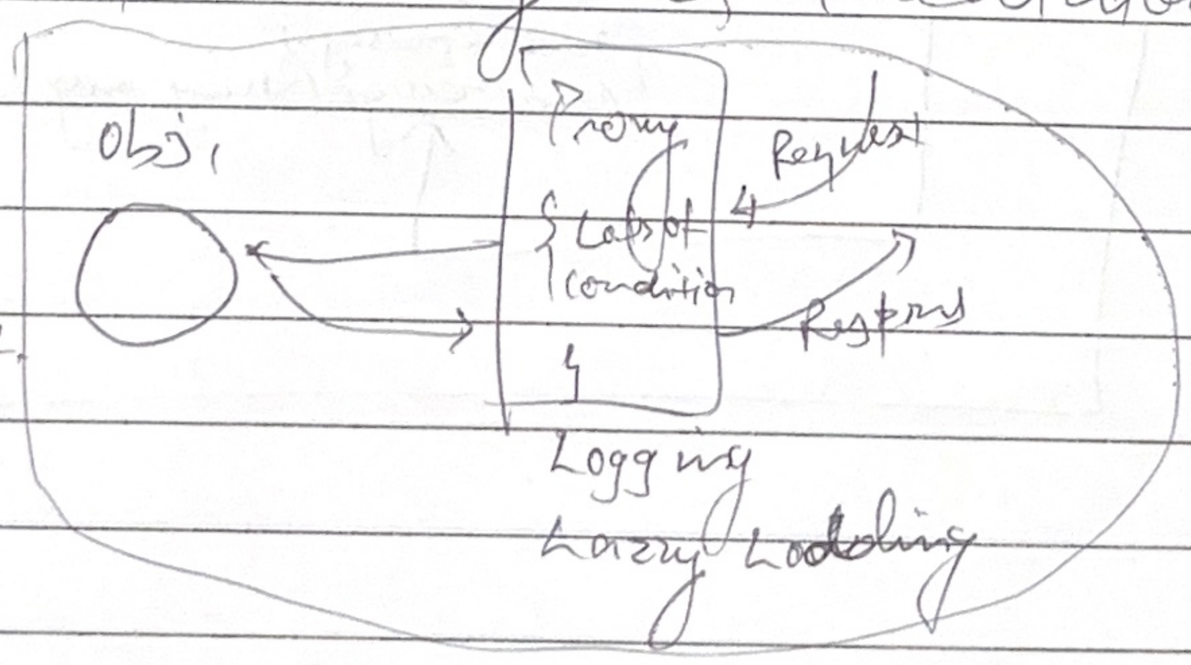
Different b/w observer & mediator:

intent is notify to all when state is changed



Different b/w Proxy & mediator:

intent is different.



(*) Null object DP.

Date: / /
Page No.:

→ It is behavioural DP. to handle null Object.

Problem: → how to handle null object.

```
private static void printVehicleDetails(Vehicle vehicle){
    System.out.println("Seating capacity :"+ vehicle.getSeatingCapacity());
    System.out.println("fuel Tank capacity :"+ vehicle.getTankcapacity());
}
```

value → Null
object

→ It will throw Null Pointer exception when vehicle object is Null.

So, Solution is, to handle NPE.

```
private static void printVehicleDetails (Vehicle vehicle){
    if( vehicle != null ){
        System.out.println("Seating capacity:" + vehicle.getSeatingCapacity());
        System.out.println("Fuel Tank capacity:" + vehicle.getFuelTankcapacity());
    }
```

→ If there are so many method in a class then at how many places we have to check the condition

unnessary checks.

So, to avoid this condition also

** "Here comes the NULL object DP".

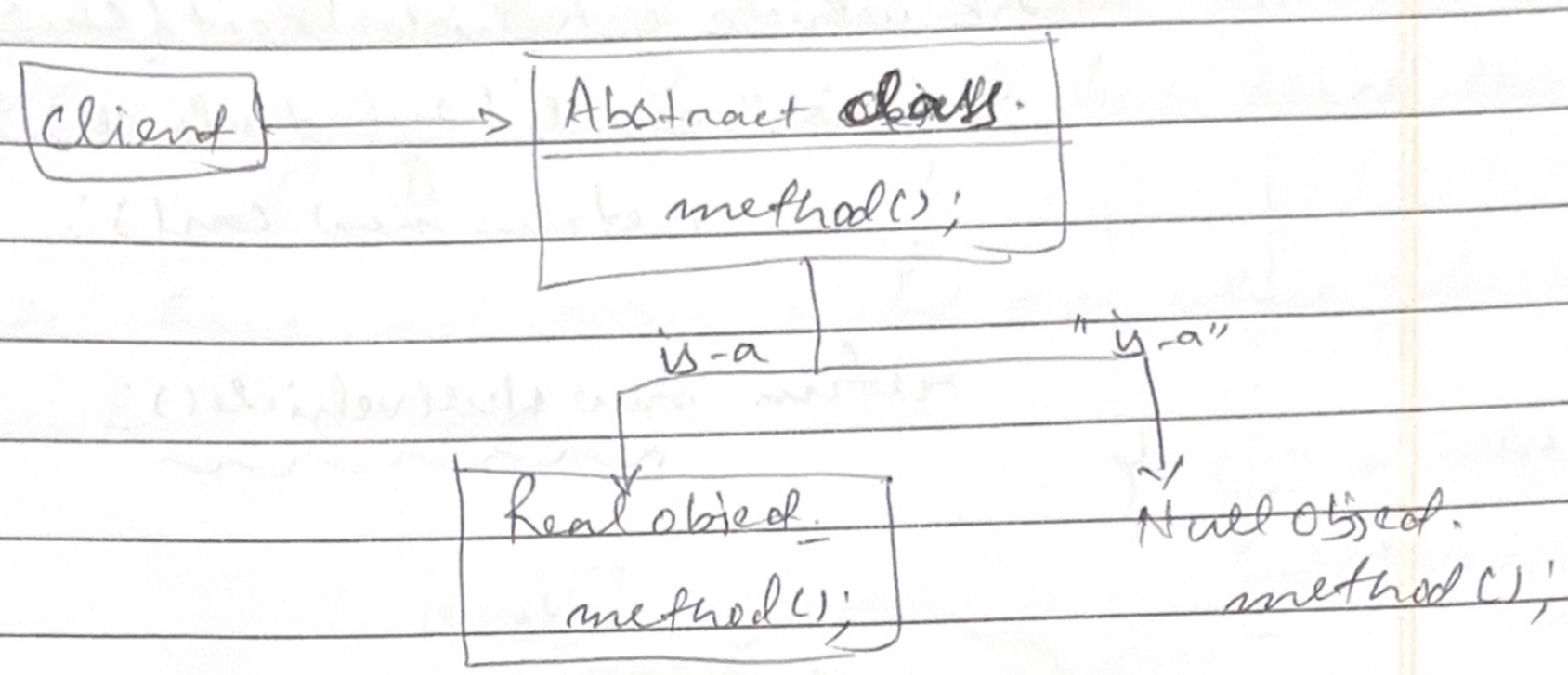
* → A null object replaces Null return type.
→ No need to put If check for checking null.
→ NULL objects reflects do nothing or default behaviour.

```
{ if - -
  {
    return vehicle.
  }
  return NULL;
}
```

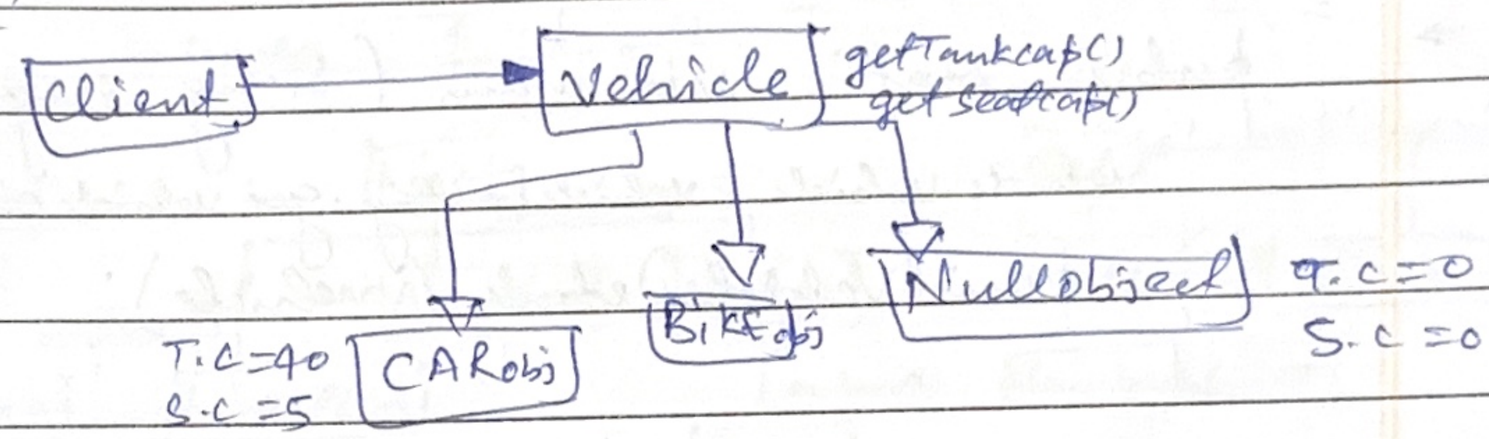
Null object.

Date: / /
Page No.:

UML of Null Object DP.



Example :




```
→ public interface Vehicle {
      int getTankcapacity();
      int getSeatcapacity();
  }

→ public class Car implements Vehicle {
    @override
      public int getTankcapacity() { return 40; }
    @override
      public int getSeatcapacity() { return 5; }
  }

→ public class NullVehicle implements Vehicle {
    @override
      public int getTankCapacity() { return 0; }
    @override
      public int getSeatcapacity { return 0; }
  }
```

Default Behavior

Date: / /
Page No.:

```
public class VehicleFactory {
    static vehicle getvehicleobject (String typeofvehicle){
        if ("car".equals (typeofvehicle)){
            return new car();
        }
        return new Nullvehicle();
    }
}
```

client code =

```
public class Main {
    public static void main (String args[]){
        vehicle vehicle = vehicleFactory.get vehicle object ("BIKE");
        printVehicleDetails (vehicle);
    }

    private static void printvehicleDetails (vehicle vehicle){
        System.out.println("Seating capacity:" + vehicle.getSeatingcapacity());
        System.out.println("fuel Tank capacity:" + vehicle.getTankcapacity());
    }
}
```

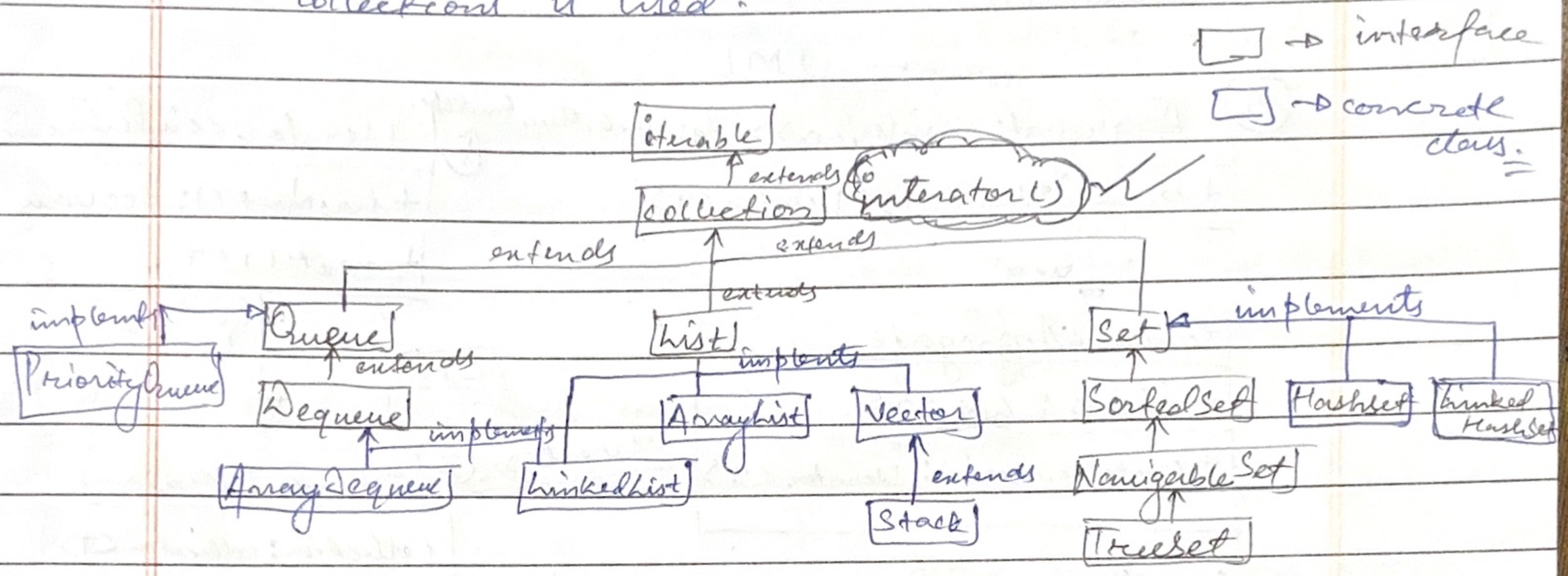
"CAR"
40
5
0,0

(xi) Iterator DP.

Date: / /
Page No.:

Iterator is a behavioural DP that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree etc.)

clients have not bother about the which type of collection is used.



Java collection.

* Each collection stores the their data on their own manner.
* So, client have not to bother about the in which manner data is retrieved from the collection.
* So, for that collection interface has an iterator() method

```
→ public class LinkedHashSetExample {
    public class void main (String args[]){
        Set <Integer> intSet = new LinkedHashSet();
                                  PQ, Stack
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
client → Iterator <Integer> iterable = intSet.iterator();
        while (iterable.hasNext()){
            int val = iterable.next();
            System.out.println(val);
        }
    }
}
```

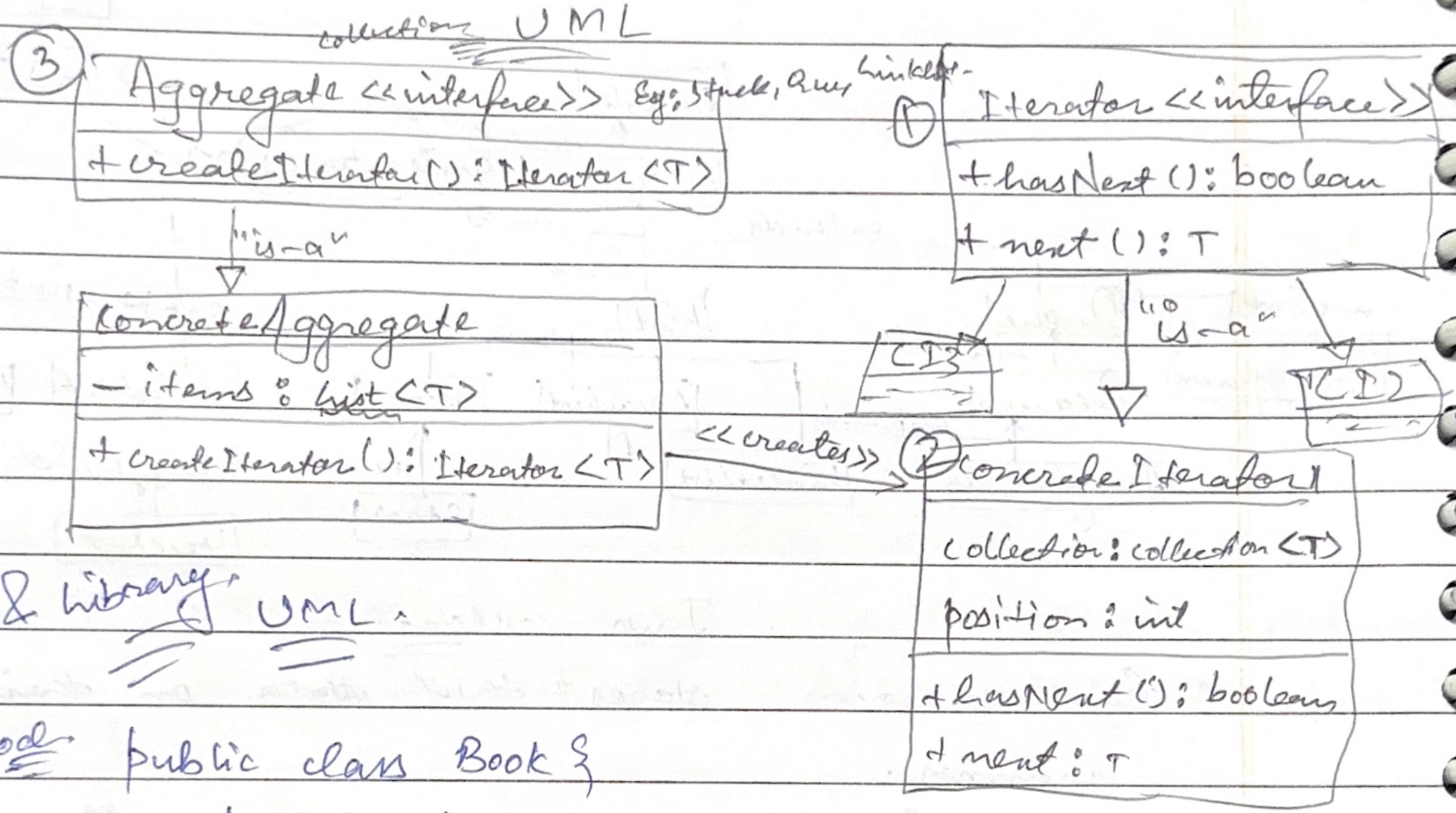
This logic remain the same

how

Date: / /
Page No.:

* How iterator method works?
→ It is a iterator DP.
→ * So, it's a behavioural DP, that provides a way to access element of a collection sequentially without exposing the underlying representation of the collection.

collection UML



Book & library UML.

code
```
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;
    }
}
```

```
public interface Aggregate {
    Iterator createIterator();
}
```

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

"is-a"

```
public class Library {
    private List <Book> bookList;
    public Library (List <Book> bookList) {
        this.bookList = booksList;
    }

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

<<creates>>

```
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++);
        }
```

"is-a"

```
public class Client {
    public static void main(String [] args) {
        Client code
        List <Book> booksList = Arrays.asList(
            new Book (100, "Science"),
            new Book (200, "Maths"),
            new Book (300, "GK"),
            new Book (400, "Drawing)
        );
        Library lib = new Library (booksList);
        Iterator iterator = lib.CreatIterator();
        while (iterator.hasNext()) {
            Book book = (Book) iterator.next();
            System.out.println (book.getBookName());
        }
    }
}
```

Creational DP.

Date: / /
Page No.:

(1) Creational Design Pattern provide various object creation mechanisms, which increase flexibility and reuse of existing code.

(2) **Factory DP:**

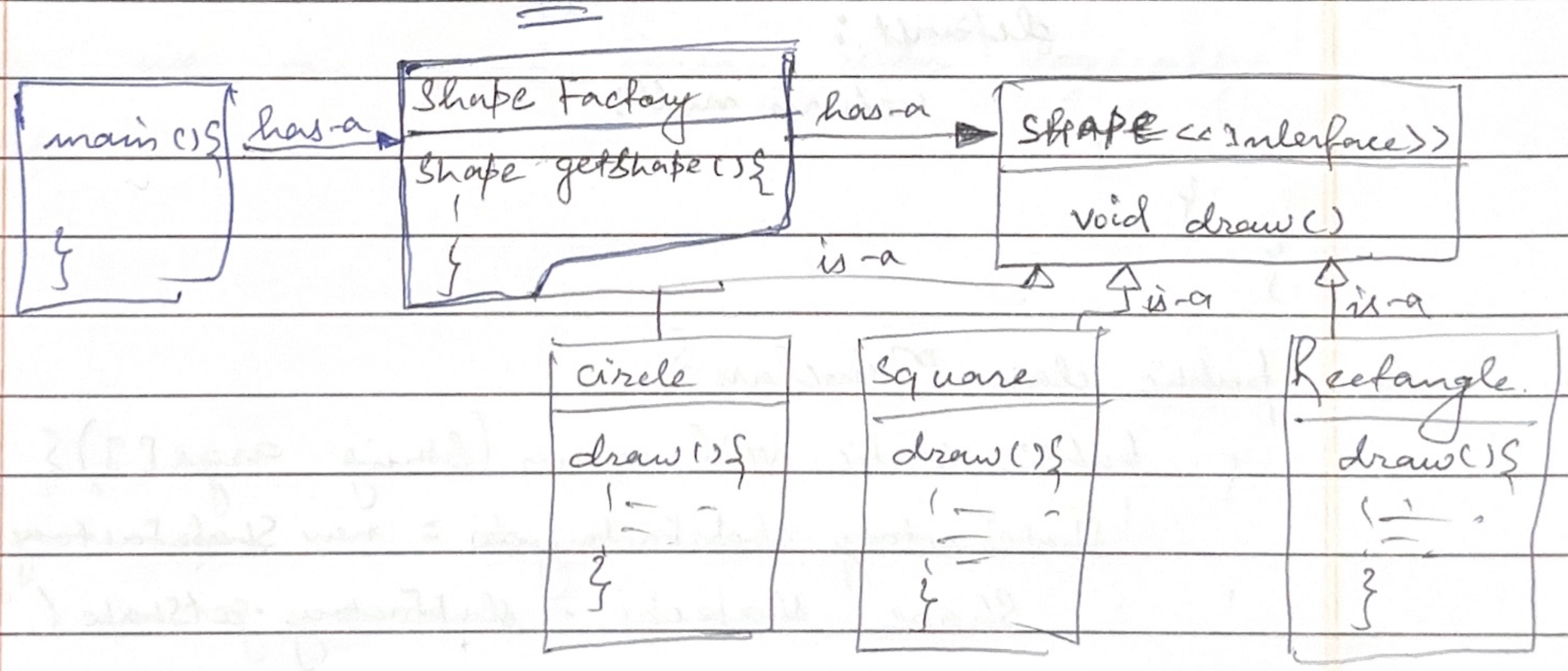
→ Factory method is also known as Virtual constructor.

→ Factory method is a creational design Pattern that provides an interface for creating objects in a superclass, but allows subclass to alter the type of objects that will be created.

conditional when condition 1 is fulfilled then create object1
" " 2 " " " " object2
" " n " " " " " n.

Problem: code will be very nasty code, riddled with conditions that switch the app behaviour.

UML.



Shape Factory hold all the objects and based on some condition it will create the objects.

Date: / /
Page No.:

→ public interface Shape {
void draw();
}

→ public class Rectangle implements Shape{
@override
public void draw(){
System.out.println("Rectangle");
}
}

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

→ public class ShapeFactory {
Shape getShape(String input){
switch(input){
case "CIRCLE":
return new circle();
case "RECTANGLE":
return new Rectangle();
default:
return null;
}
}
}

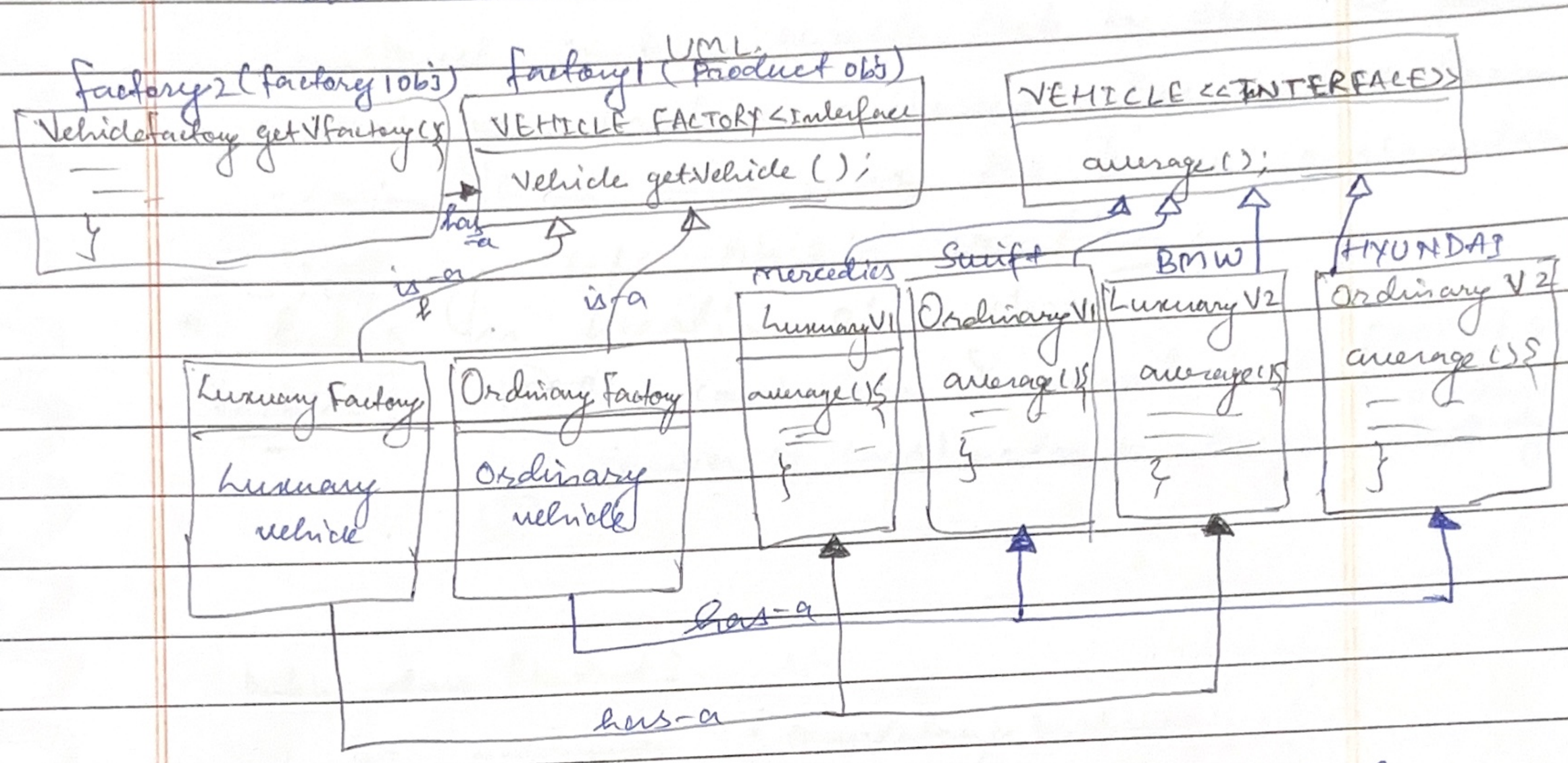
→ public class MainClass {
public static void main(String args[]){
ShapeFactory shapeFactoryobj = new ShapeFactory();
Shape shapeobj = shapFactory.getShape("CIRCLE");
shapeobj.draw();
}
}

→ why need of factory DP?
* agar humne objects create karna h or bahut badda code repo h, toh ho skta a hume same object bahut jagah create karna padde on basis of some condition so there would be code duplicacy arises.

(22) Abstract Factory DP.

Date: / /
Page No.:

→ Abstract Factory is a factory of factories.
→ It is a creational DP that lets you produce families of related objects without specifying their concrete classes.



→ When more than one factory is needed:
→ We can group factory into logically.

No code.

(iii) Builder Design Pattern

Date: / /
Page No.:

Problem & How?
StringBuilder?
Builder Vs Decorator

→ Builder DP is a creational DP that lets you construct complex objects step by step. The pattern allows you produce different types and representation of an object using the same construction code.

→ Steps by steps Objects creation.
→ helps in creation of an object.

Problem:- (i) Big constructor Parameter List. (bcz of so many optional property)
(ii) Number of constructor is huge.
(iii)

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

So many optional property:
so, how it's constructor gonna look's like.

```
    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;
    }
}
```

Date: / /
Page No.:

we know only rollNumber is mandatory.

Let's say Now we will construct constructors like below:-

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

```
public Student (int rollNumber, String fatherName, String motherName){
    this.rollNumber = rollNumber;
    this.motherName = motherName;
    this.fatherName = fatherName;
}
```

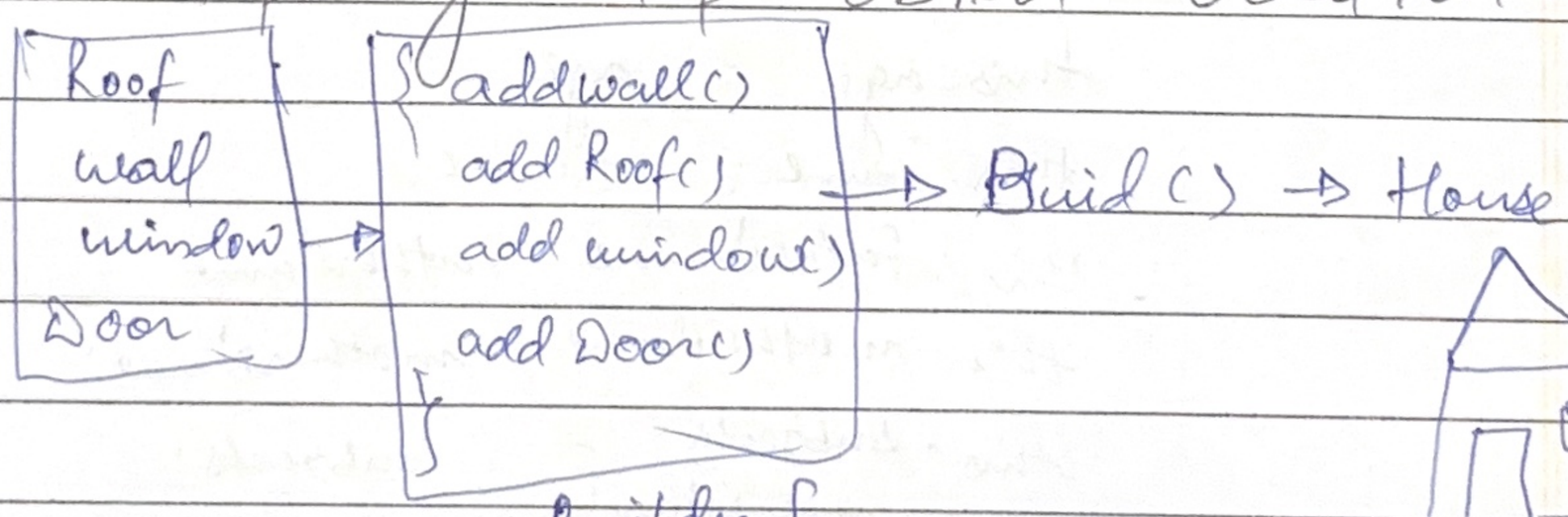
(III) Problem throws error bcz same signature of parameter in constructor

```
public Student (int rollNumber, String fatherName, String Name){
    this.rollNumber = rollNumber;
    this.fatherName = fatherName;
    this.name = name;
}
```

So, lots of Problem comes due to lots of Optionals property. So, here builder OP comes into the picture.

Sol^n

Step by Step object creation happens in BOP.



Builder form
mediator form

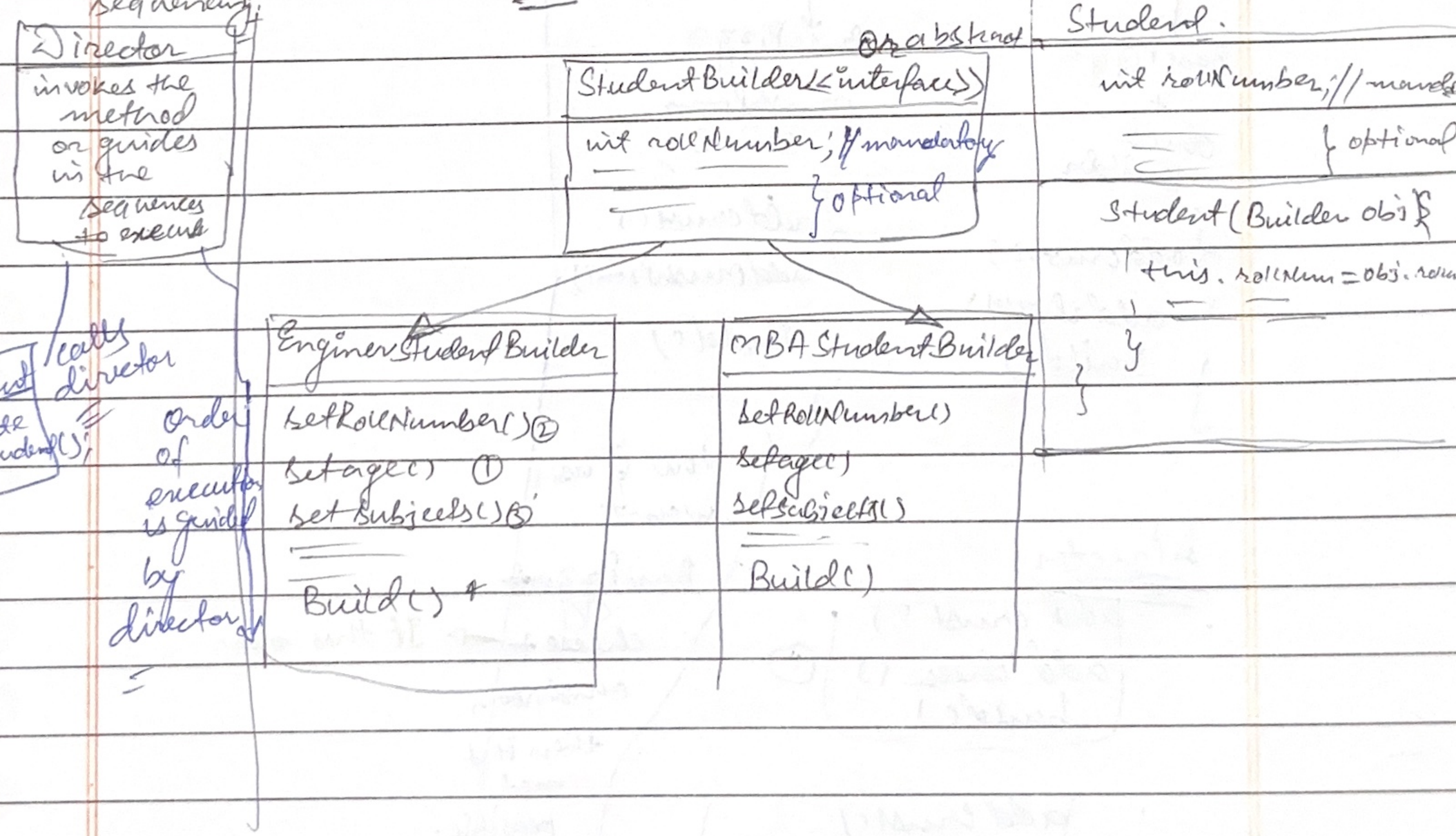
objects, To create a big objects.

```
public class Student {
    int rollNumber;   // mandatory
    int age;
    String name;
    String fatherName;
    String motherName;
    List<String> subjects;
    public Student (StudentBuilder builder){
        this.rollNumber = builder.rollNumber;
        this.age = builder.age
        this.fatherName = builder.fatherName;
        this.motherName = builder.motherName;
        this.subjects = builder.subjects;
    }
}
```

} optional

⇒ StudentBuilder have all the ~~parameters~~ property

UML.



Date: / /
Page No.:

Builder DP

→ Creation DP

→ Responsible for creation of an object.

Pizzza
{ Base crust // mandatory
cheese cheese ;
mushroom mushroom ; } optional
}

<< interface >>

PizzaBuilder.

BasePizzza + cheese Builder
{ addcrust()
addcheese()
Build()
}

BasePizzza + mushroom Builder
{ addcrust()
addmushroom();
Build()
}

Director:

- add crust()
add cheese()
build() ①

add crust()
add mushroom()
build() ②

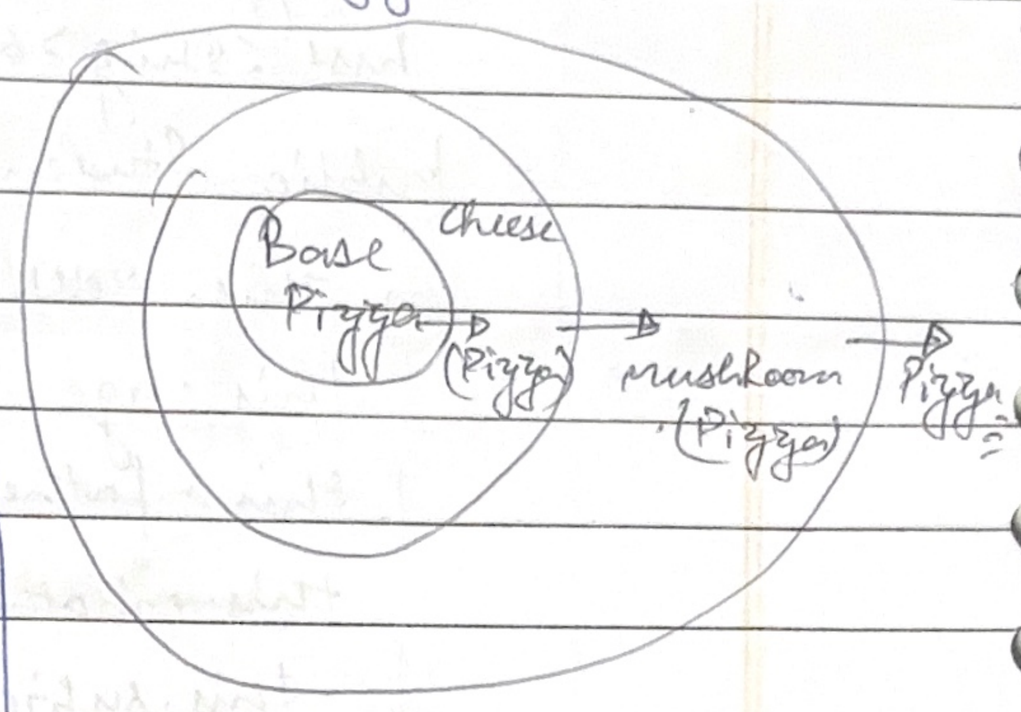
Now if user want BasePizzza + cheese + mushroom → If this then it is not possible, in Builder Design Pattern
↳ Not handle Dynamic Requests

Decorator DP

→ Structural DP

→ helps you in creating complex objects. which is small small objects.

Pizzza



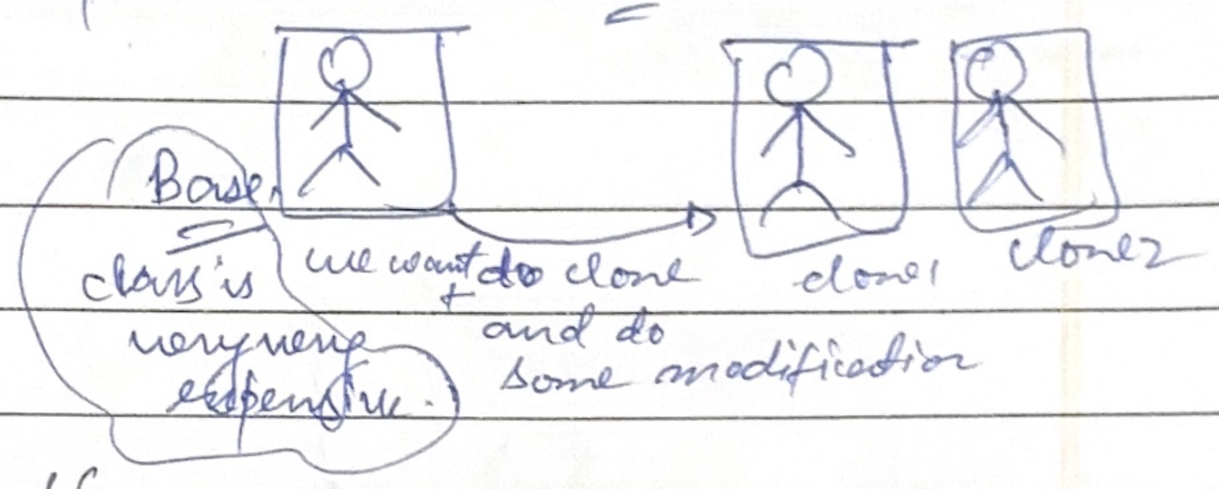
Supports Dynamic Nature of an objects.

(iv) Prototype DP - (clone)

Date: / /
Page No.:

- Also known as clone
- Prototype is a creational DP that lets you copy existing objects without making your code dependent on their classes.
- Creational DP responsibility is to create objects / controls the creation of objects.
- It is used when we have to make copy/clone from existing objects.

Example :- Expensive Robots.



Let's understand the problem first.

Problem :-

```
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;
    }
}

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

private members in Analog so, error.

Date: / /
Page No.

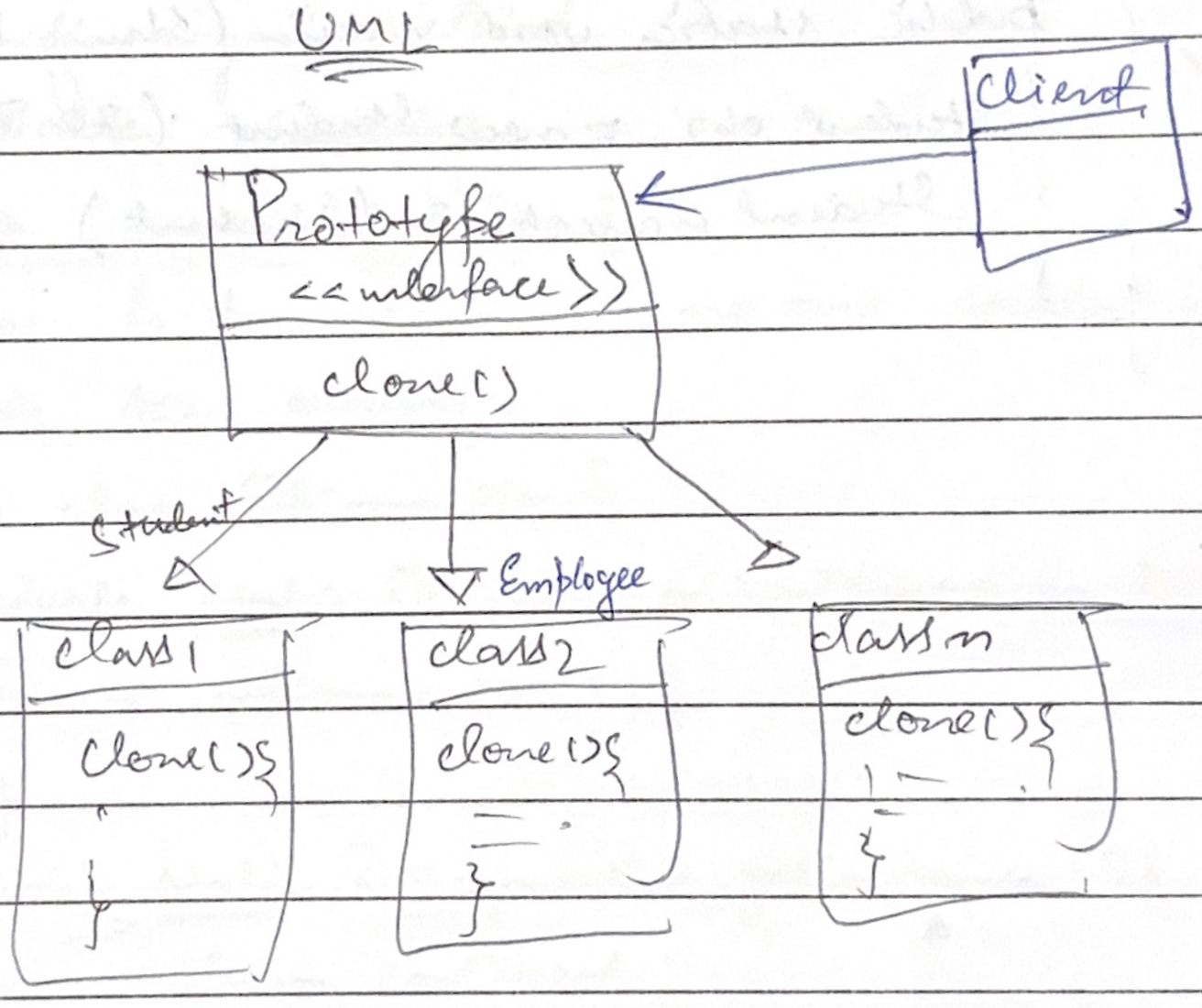
→ So, suppose getrollNumber() method in Student is also private So, how we will access this?

Problem:
→ (i) There is certain private member then while cloning how you will clone those private members?

→ (ii) client has to know all the members while cloning one by one: 98 members have to copy
2 " " to leave.
out of 100:

Solution :-
It is the cloning responsibility of the class itself. not the client.

UML



Example code:

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

public class Student implements Prototype{
    int age;
    private int rollNumber;
    String name;
    Student(){
    }
```

Date : / /
Page No. :

```
    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);
    }
}

→ Public class main(){
    public static void main(String[] args){
        Student obj = new Student(20, 75, age);
        Student cloneobj = (Student) obj.clone();
    }
}
```

(V) Singleton Design Pattern

Date: / /
Page No.:

→ It is a creational DP that lets you ensure that a class has only one instance, while providing a global access point to this instance.

→ It is mostly used in dependency injection

→ It is used in DB connection (ATM) (only 1 connection needed)

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

4 ways to achive this :-

(a) Eager Initialization
(b) Lazy Initialization.
(c) Synchronized method
(d) Double checking

(a) Eager Initialization :-

Constructor is private so no one calls from outside to create the objects.

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

So, called eager bcz. it is initialized early due to static. So, only one object is created when java class is run then only 1 object is created. Does not matter you are using it or not. linked to class not object.

method to expose to create new object.

→ client:

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

} } ← return conObject;

Date: / /
Page No.:

(b) Lazy Initialization :-

when getInstance method is called then at that time only we are initializing the object and creating only when object is null.

~~Problem~~ :-

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

Not initialized, here it is null.

Thread T_1

Thread T_2

Problem :- Suppose Two thread T_1, T_2 comes simultaneously at the same time and then both checks conobject are null yes, it is null then here two different objects are created. but in singleton we have to create only one object.

Solution is synchronized

(2) Synchronized method.

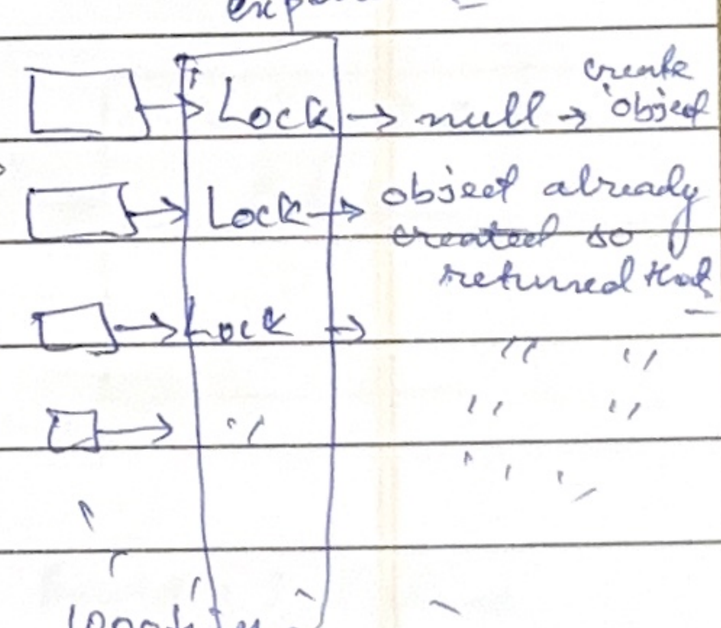
↳ Allow only one thread to pass or check when Thread T_1 doing execution till then Thread T_2 is waiting.

Date: / /
Page No.:

Problem:- But synchronized method is too much expensive. So, suppose in my codebase at 100 places we are calling getInstance() then Entire till lock for 100 times so, it is too much expensive.

Solution is Double locking.

```
public class DBconnection {
    private static DBconnection conObject;
    private DBConnection () {
    }
                                    // synchronized method
Thread T1 go → synchronized public static DBConnection getInstance () {
Thread T2 will wait till T1 completes its execution.
        if ( conObject == null ) {
            conObject = new DBConnection ();
        }
        return conObject;
    }
}
```



** Synchronized method is more expensive than synchronized block.

(d) **Double checking :-** It puts two times condition checking so less expensive.

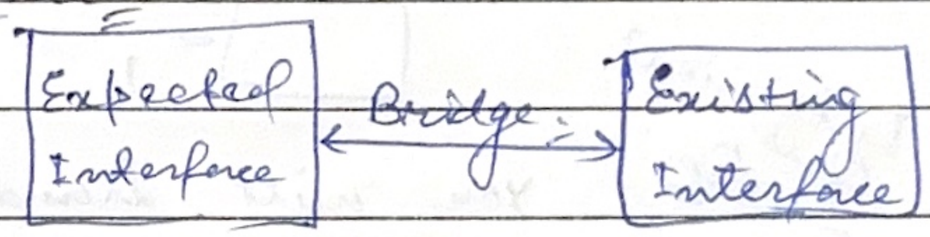
```
public class DBConnection {
    private static DBConnection conObject;
    private DBConnection () {
    }
Thread T1 goes public static DBConnection getInstance () {
Thread T2 goes, both goes here
        if ( conObject == null ) {          // 1st check
            T1 goes synchronized ( DBConnection.class ) {   // synchronized block.
T2 will wait till T1 completes
                if ( conObject == null ) {  // 2nd check.
                    conObject = new DBConnection ();
                }
            }
        }
        return conObject;
    }
}
```

Structural Design Pattern

These design Patterns ~~provide~~ deal with class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionality.

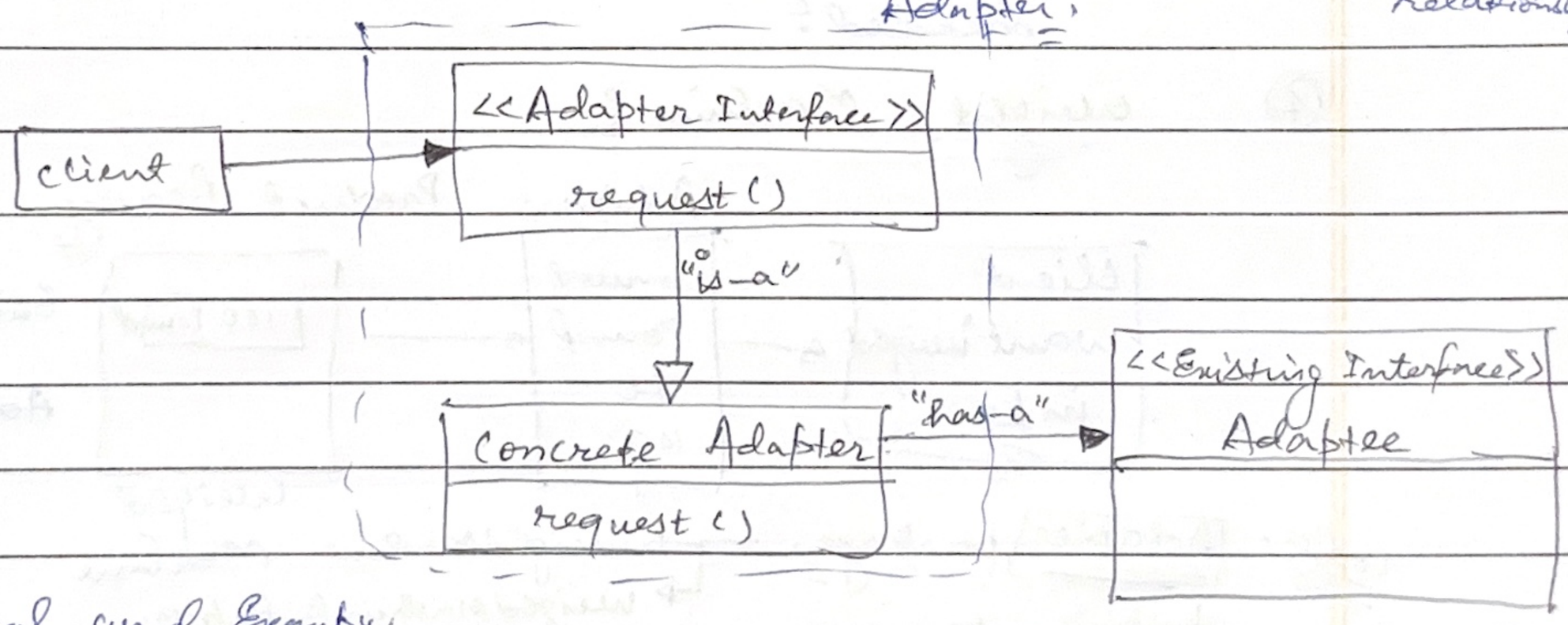
(i) Adapter Design Pattern (Wrapper)

- Also known as Wrapper.
- It is Structural DP that allows objects with incompatible interfaces to collaborate.
- It bridge the gap between existing interface and expected interface.



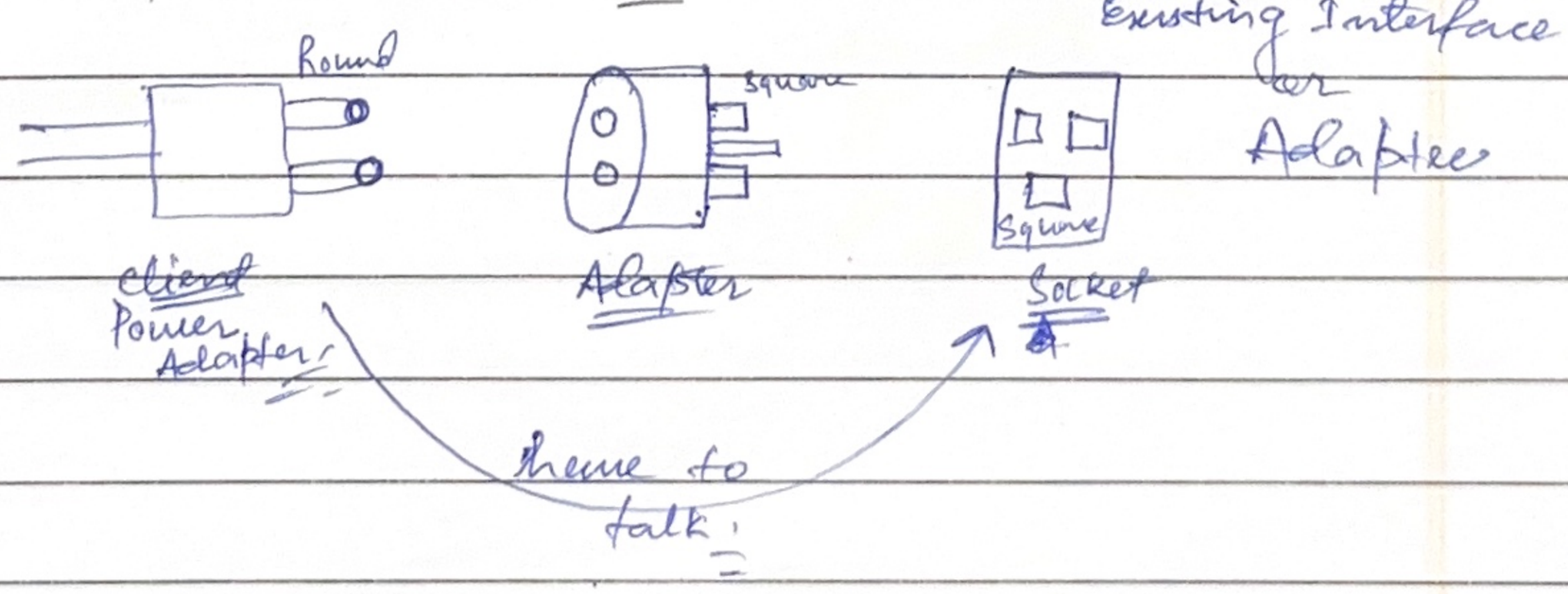
UML (combination of "is-a" and "has-a" relationship)

Adapter.



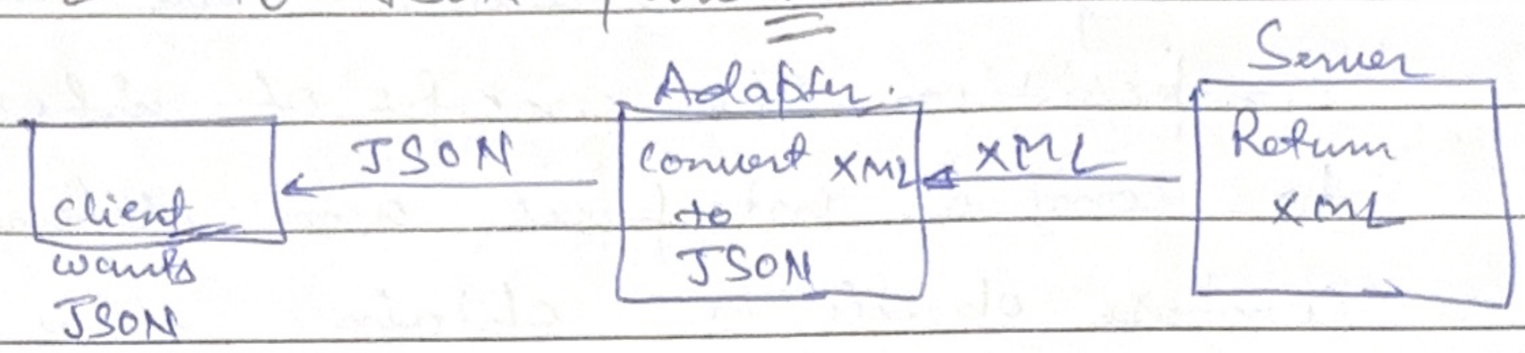
Real world Example:

(i) PowerAdapter & Socket.



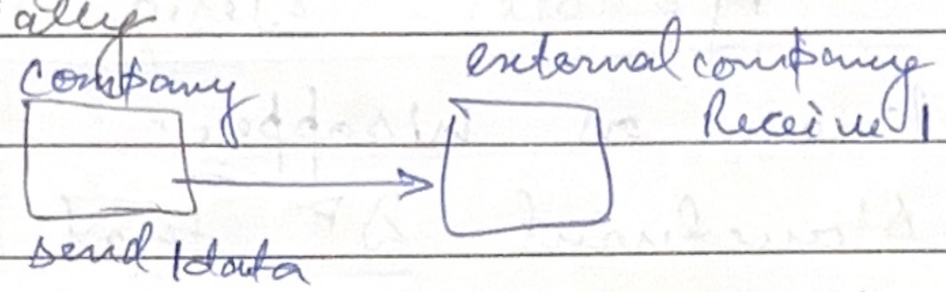
Date : / /
Page No. :

② XML to JSON parser:



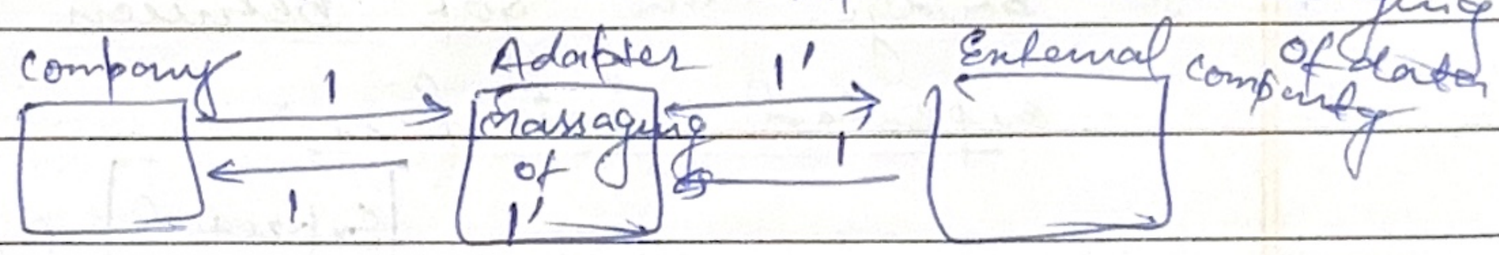
③ company & External company:

** Initially



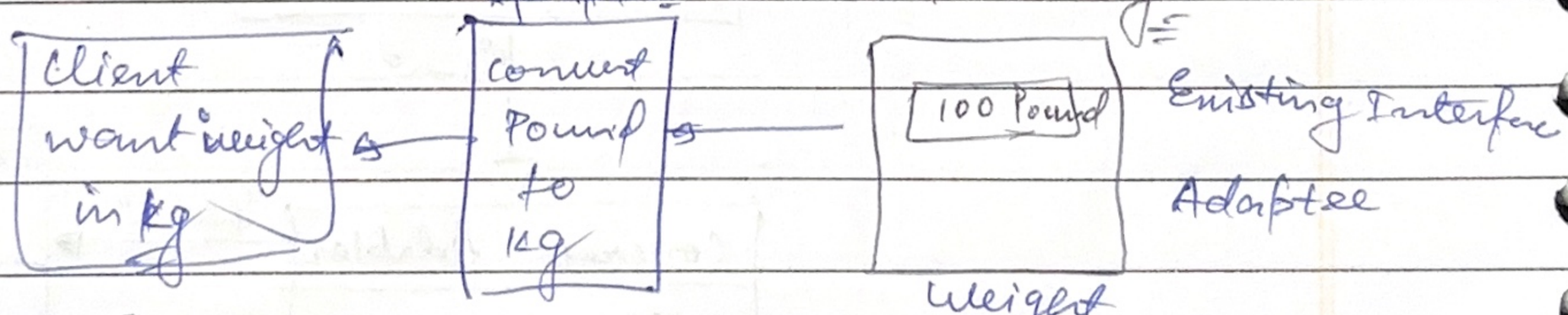
but now Ext. company want 1' when company send 1 data

So, in b/w we need Adapter to do massaging of data



Interview Qns: How you will handle data when you want to send or require a request that are different than needed?

④ Weight Machine :-



code: Adaptee package: → WeightMachine
→ WeightMachineForBabies.

public m

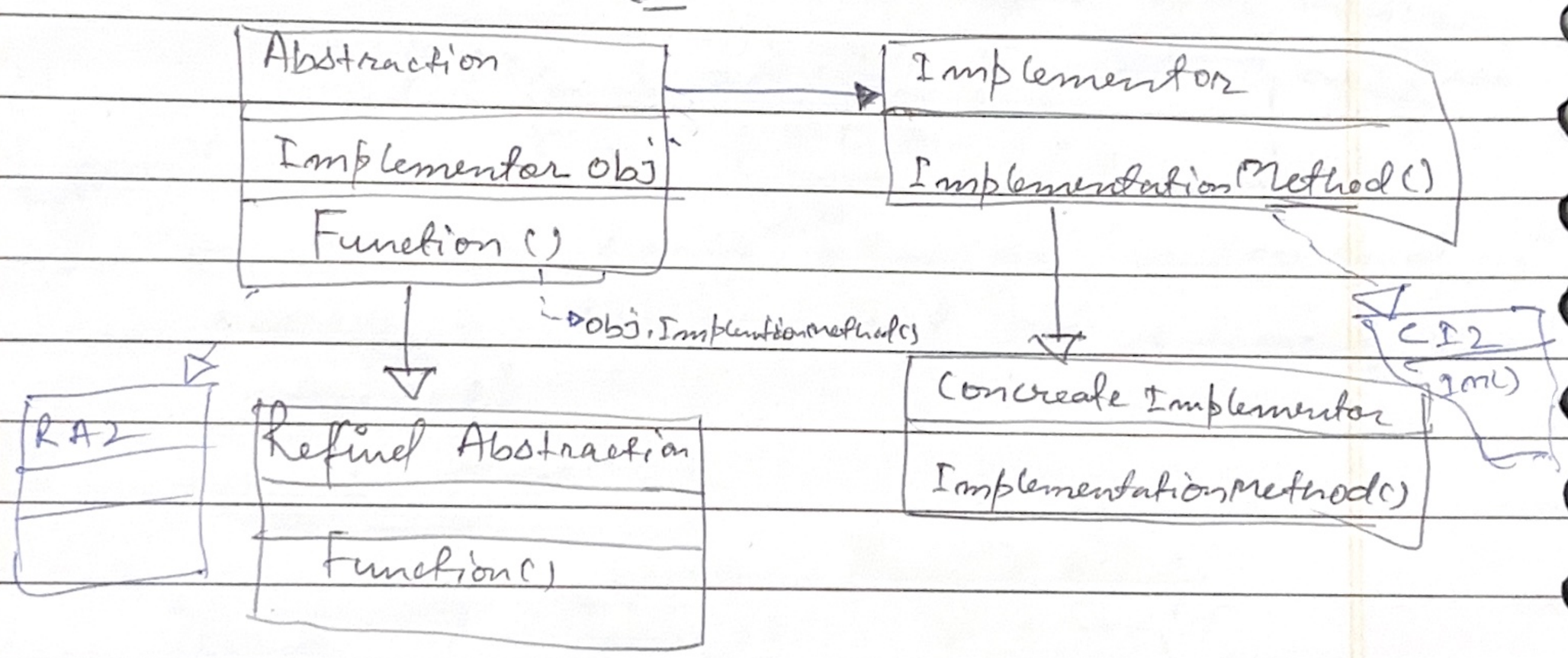
(21) Bridge Design Pattern

Date: / /
Page No.:

→ It is structural DP that lets you split a large class or a set of closely related classes into two separate hierarchies - abstraction and implementation - which can be developed independently of each other.

→ "Bridge design Pattern decouples an abstraction from its implementation so that the two can vary independently."

UML




```
public abstract class LivingThings {
    abstract public void breatheProcess();
}
```

Tightly coupled

```
public class Dog extends LivingThings
    @override
    public void breatheProcess{
        //Breathe through Nose.
        //Inhale Oxygen from Air
        //Exhale CO2
    }
}
```

Tightly coupled

```
public class Fish extend LivingThings
    @override
    public void breatheProcess{
        // breathe through GILLS
        // inhale O2 from water
        // Exhale CO2
    }
}
```

Tightly coupled without order Bird

```
public class Tree extends LivingThings
    @override
    public void breatheProcess{
        // breathe through LEAVES
        // Inhale CO2
        // Exhale O2 through photo synthesis
    }
}
```

Date : / /
Page No. :

Problem :- It is tightly coupled with abstraction.

suppose, we want to introduce a new breatheProcess for Bird but we cannot add a newbreatheProcess until and unless we creating a new child of LivingThings bcz it's tightly coupled.

```
Bird
@Override
public void breatheProcess{
// Inhale through Nards
//   "   O2
// Exhale CO2
```

Solution is Bridge DP.

```
Public class interface BreatheImplementor{
    public void breathe();
}
```

```
public class LandBreatheImplementation
implements BreatheImplementor{
@override
    public void breathe(){
    //Breath through NOSE
    //Inhale O2 from Air
    // Exhale CO2
    }
}
```

```
public class WaterBreatheImplementation
implements BreatheImplementor{
@override
    public void breathe(){
    // Breath through GILLS
    //Absorb O2 from H2O
    // Release CO2
    }
}
```

```
public class TreeBreatheImplementation
implements BreatheImplementor
{
@override
    public void breathe(){
    // breath through LEAVES
    // inhale CO2
    // Exhale O2 through Photosynthesis
    }
}
```

```
XYZBreathe
```

Date: / /
Page No.:

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

```
public class Dog extends
                LivingThings{
    public Dog(BreatheImplementor
                breathImplementor){
        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();
    }
}
```

How client going to invoke for example:-

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

* Here both abstraction (LivingThings) & implementor (BreatheImplementor), grow or vary independently means more child class.

```
LivingThings treeObject = new Tree(new TreeBreatheImplementation());
treeobject.breatheProcess();
```

XYZ breathe

Date: / /
Page No.:

Difference b/w Strategy & Bridge. DP:

* intent of Strategy DP is change the behaviour of content at runtime or dynamically.

* intent of Bridge DP is both content (Abstractor) & Strategy (Implementer) grow independently. Separate the logic

Composite DP -

Date: / /
Page No.:

Ex: → Calculator
→ File System.

→ It is also known as Object tree.

→ It is structural DP that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

→ It is object inside objects, happens.

Example:-

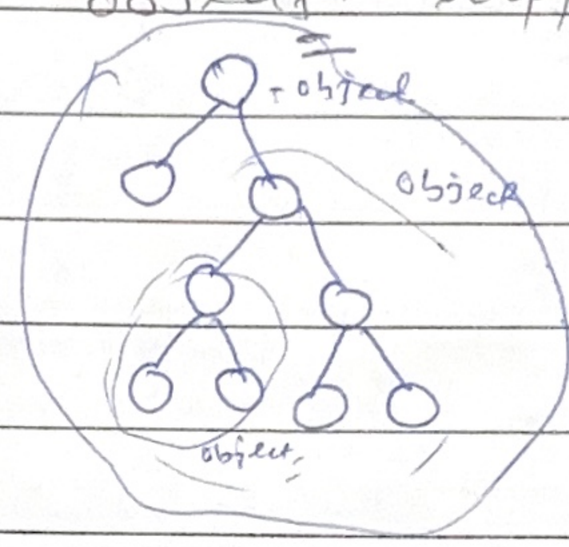
Employee

CEO

Director

Manager Marketing Team

IT Engineer



Example Delivery box.

Product Box

Product Box

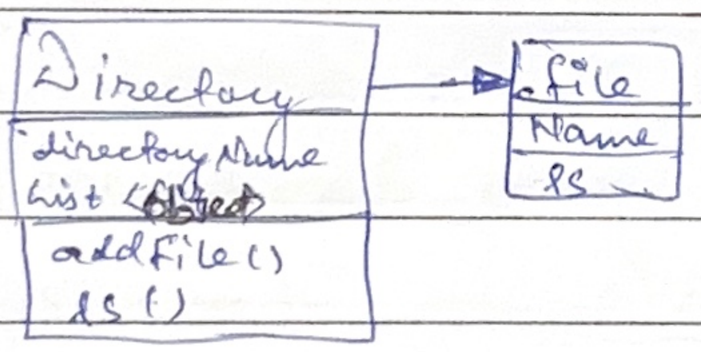
Tree like structure then use composite DP.

File System -

File or Director

(Files or Directory)

Files Directory



Problem:-

```
public class File {
    String fileName;
    public File (String fileName){
        this.fileName = fileName;
    }
    public void ls (){
        System.out.println("file name: " + fileName);
    }
}
```

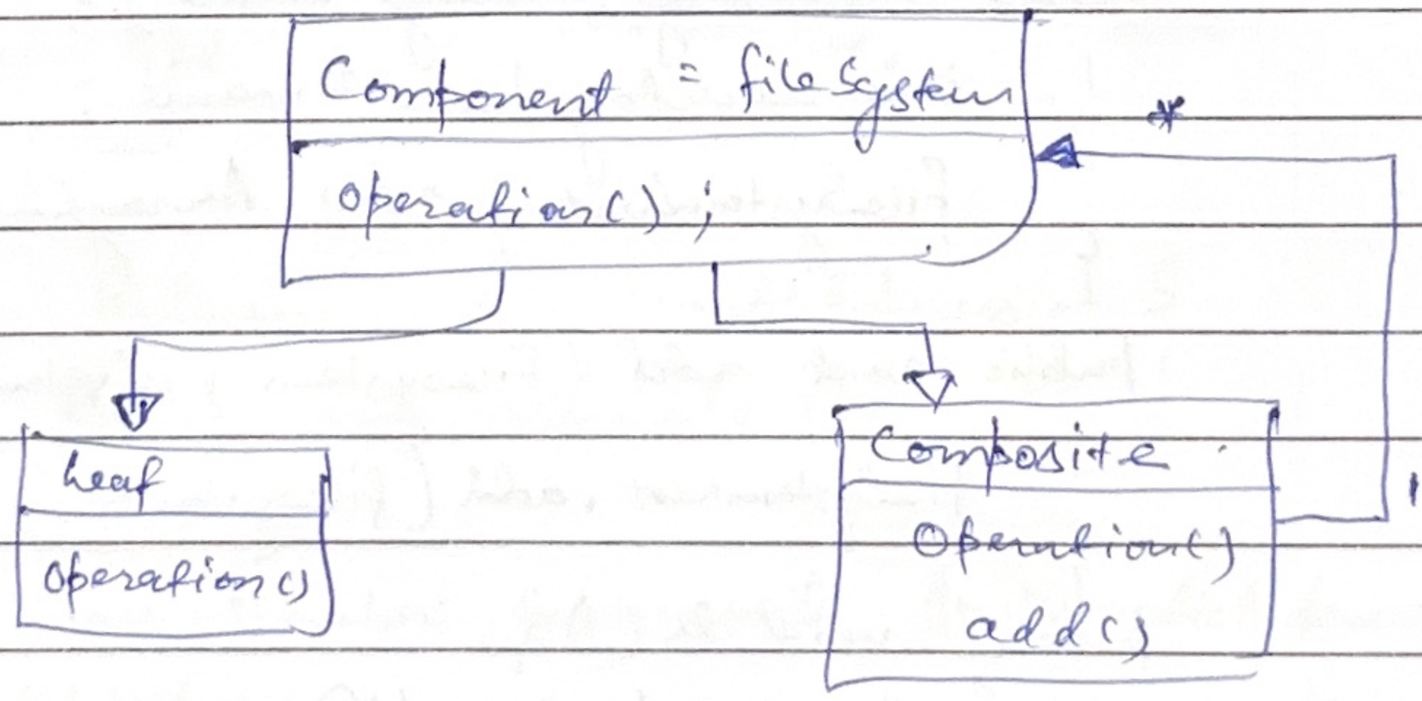
Date : / /
Page No. :

```
→ public class Directory {
    String directoryName;
    list<object> objectlist;
    public Directory (String name){
      this.directoryName = name;
      objlist = 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();
        }
      }
    }
}
```

typecasting

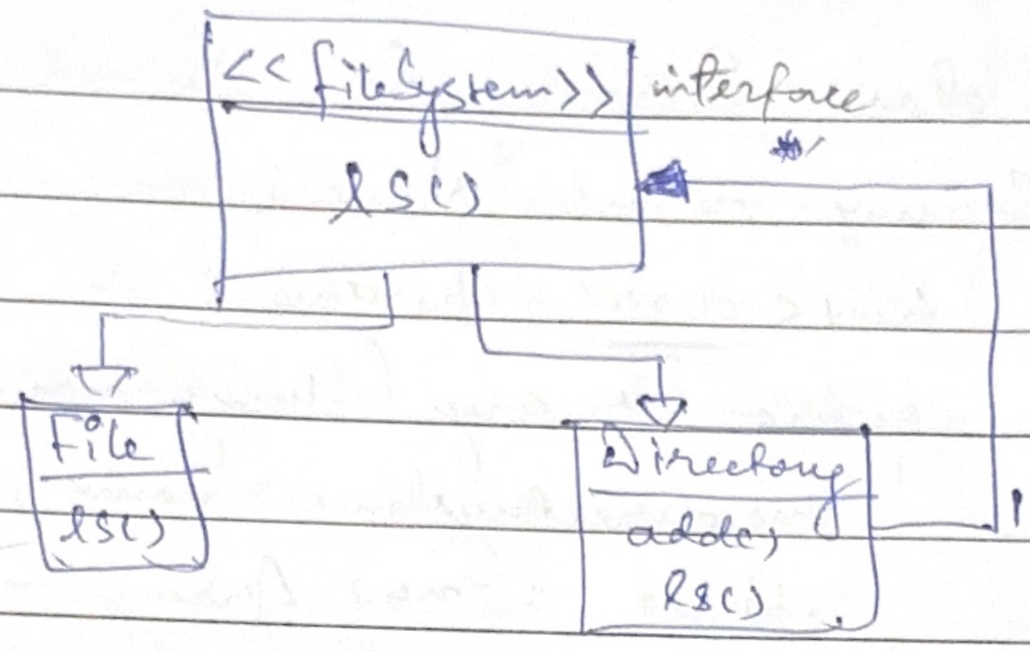
typecasting

Problem is, we have to put instanceof and typecast everytime



UML of composite DP.

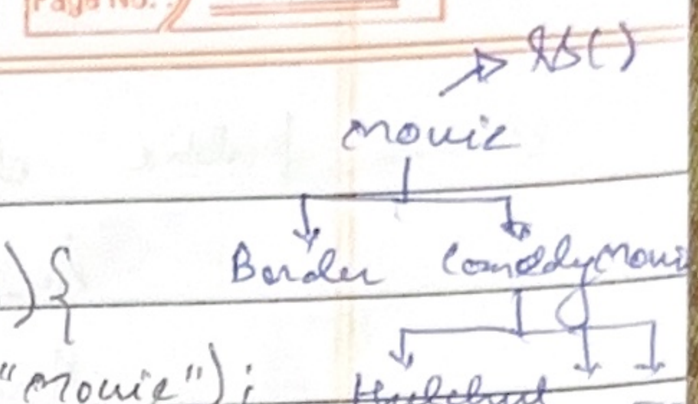
Date : / /
Page No. :



UML of FileSystem

```
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()
        }
    }
}
```

Date : / /
Page No. :

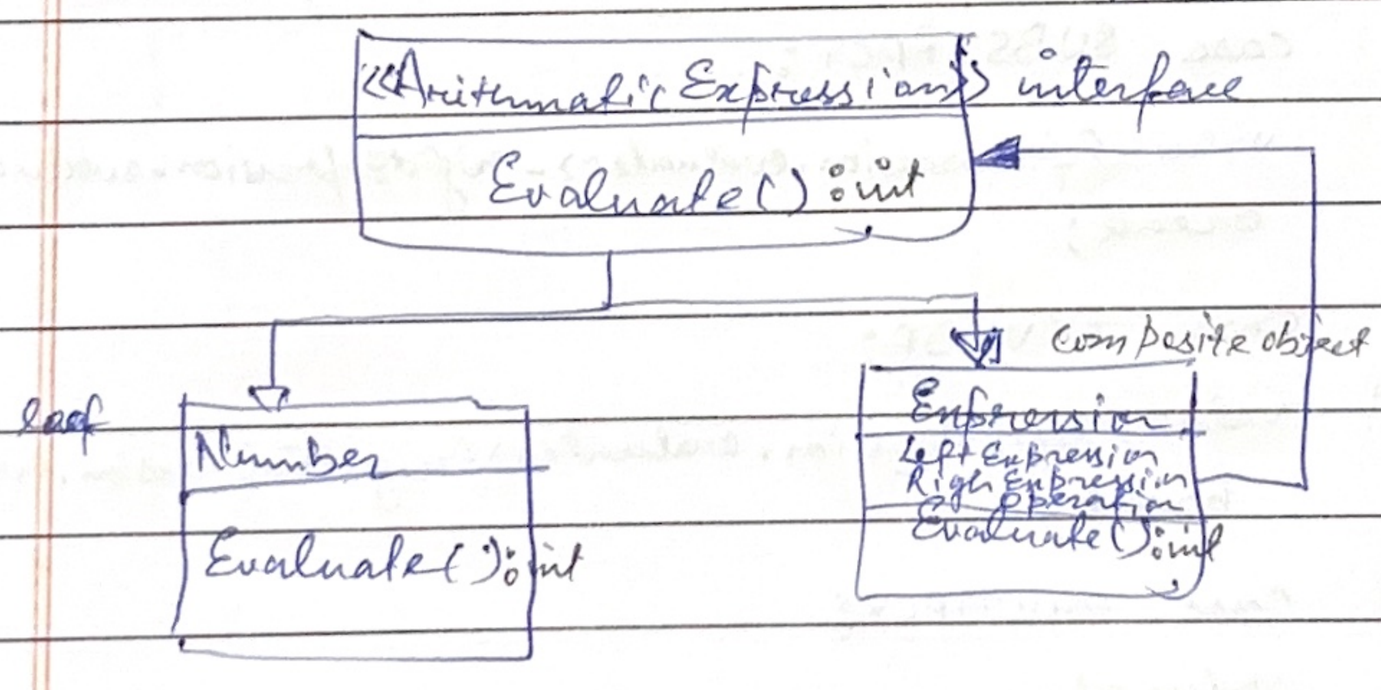


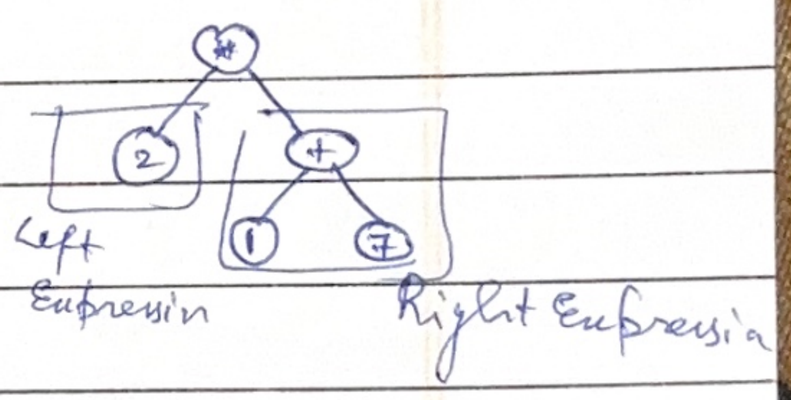

```
public class Main {
    public static void main (String [] args) {
        Directory movieDirectory = new Directory ("Movie");
        FileSystem border = new File ("Border");
        movieDirectory.add (border);

        Directory comedyMovieDirectory = new Directory ("comedyMovie");
        File hulchul = new File ("Hulchul");
        comedyMovieDirectory.add (hulchul);
        movieDirectory.add ( comedyMovieDirectory );
        movieDirectory.ls();
    }
}
```

→ DirectoryName movie
file Name Border.
Directory name Comedy movie
file Name hulchul

Example 2. Calculator with Arithmatic Calculation Expression:

2*(1+7)





UML of calculator

```
public interface ArithmaticExpression {
    public int evaluate()
}

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

Date: / /
Page No.:

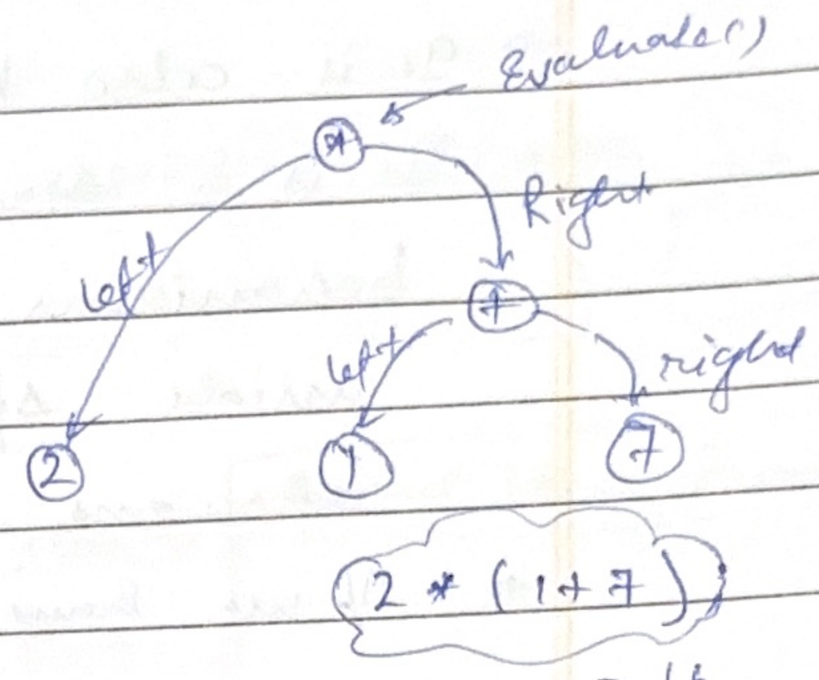
→ public class Expression implements ArithmaticExpression {
ArithmaticExpression leftExpression;
ArithmaticExpression rightExpression;
Operation operation;

public Expression (ArithmaticExpression leftPart, ArithmaticExpression 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 SUBSTRACT:
value = leftExpression.evaluate() - rightExpression.evaluate();
break;
case DIVIDE:
value = leftExpression.evaluate()/rigetExpression.evalue();
break;
case MULTIPLY:
value = leftExpression.evaluate() * rightExpression.evaluate();
break;
}
System.out.println("Expression value is: " + value);
return value;
}

Date: / /
Page No.:

→ Public enum Operation {
    ADD,
    SUBSTRACT,
    MULTIPLY,
    DIVIDE;
}



→ public class main {
    public static void main(String [] args) {
        ArithmaticExpression two = new Number (2);
        ArithmaticExpression one = new Numbe(1);
        ArithmaticExpression seven = new Number (7);
        ArithmaticExpression addExpression = new Expression(one, seven, Operation.ADD);
        ArithmaticExpression parentExpression = new Expression(two, addExpression, Operation.~~ADD~~ MULTIPLY);
        System.out.println(parentExpression.evaluate());
    }
}

(iv) Decorator DP.

Date: / /
Page No.:

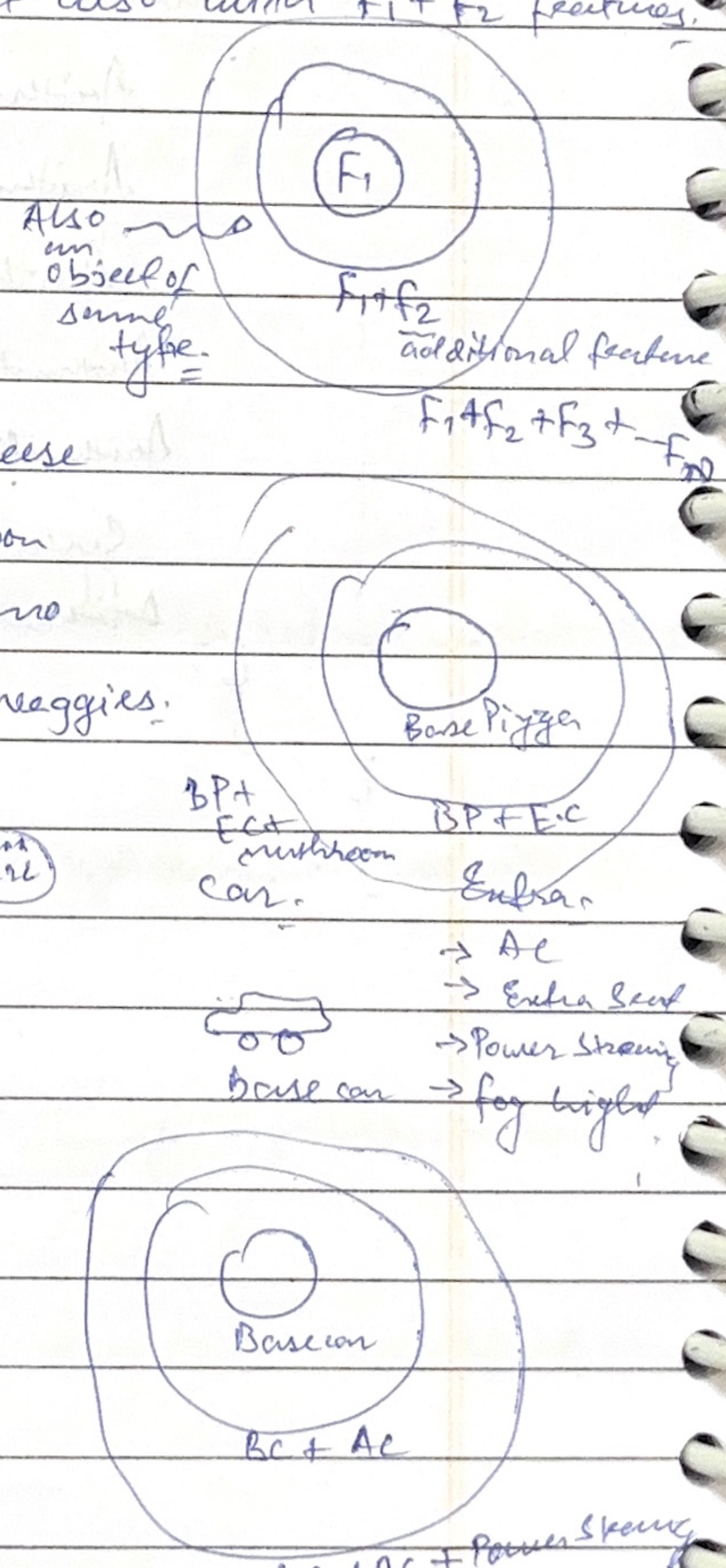
→ It is also known as Wrapper DP.

→ It is a structural DP that lets you attach new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

→ If we have Base class object having feature F_1 then wrapper class add some addition feature F_2 and wrapper whole becomes an object also with $F_1 + F_2$ features.

→ Use Case :-

(2) Pizza Shop.



Topping.
Extra cheese
mushroom
Jalepeno
Extra veggies.

(22) Coffee Shop.

Machine.
Toppings.
Cream.
Extra milk

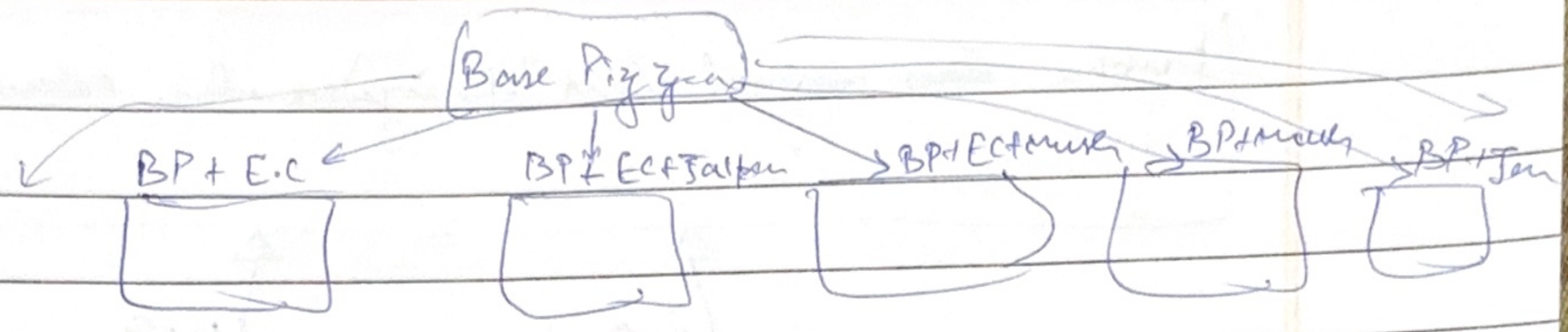
Base coffee
Decafe
Expresso.

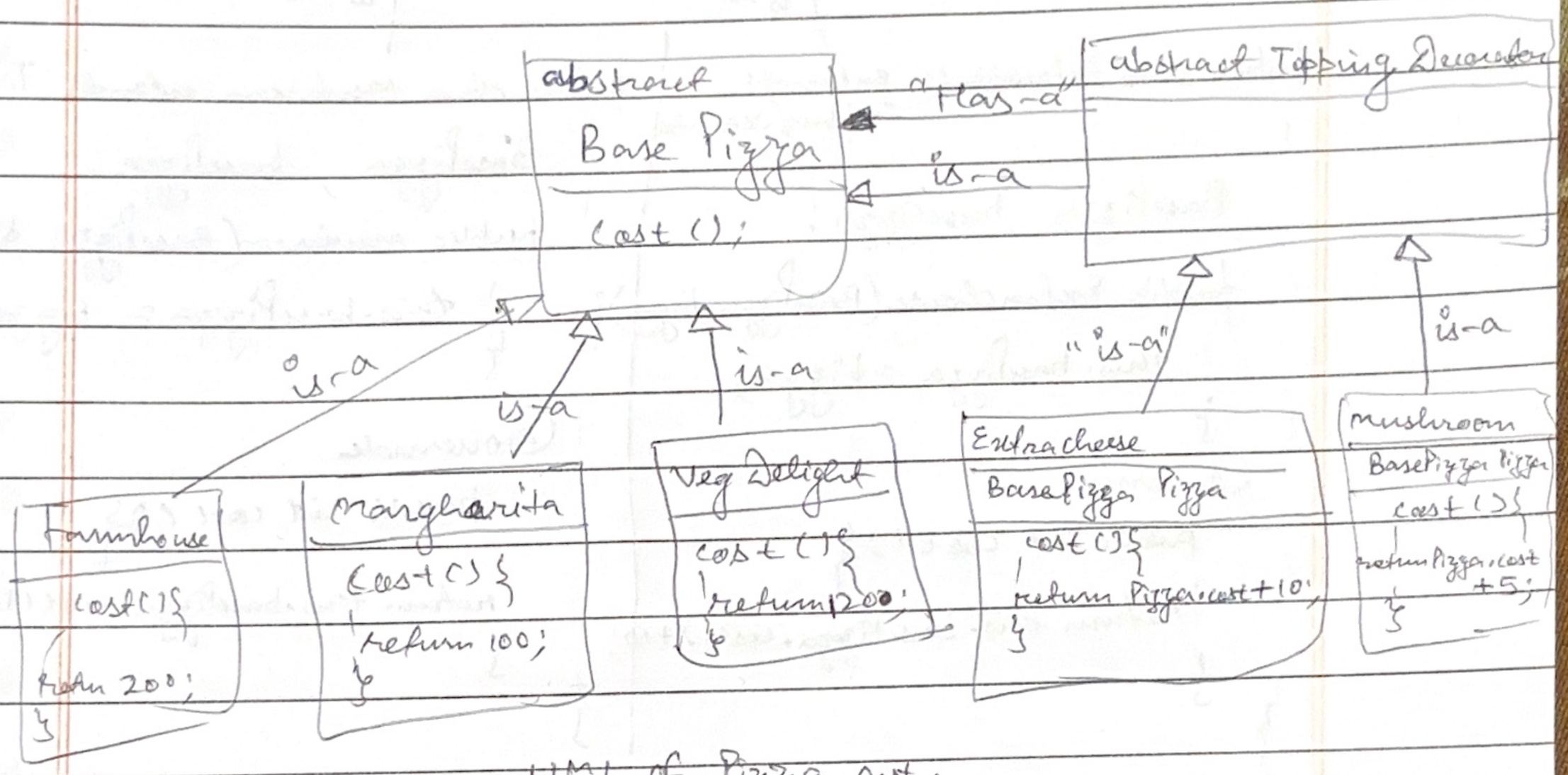
Car.
Extra.
→ AC
→ Extra Seat
→ Power Steering
→ fog light

* Why we need Decorator Pattern?

→ To avoid from class explosion.

when there are so many child class, so there to manage the whole class is difficult so we need DDP.





UML of Pizza Cost.

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

```
public class Farmhouse extends BasePizza {
    @override
    public int cost() {
        return 200;
    }
}
```

```
public class Margherita extends BasePizza
    @override
    public int cost() {
        return 100;
    }
}
```

```
public class VegDelight extends BasePizza
    @override
    public int cost() {
        return 120;
    }
}
```

Date : / /
Page No. :

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

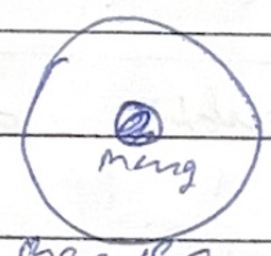
is-a

```
public class Extracheese extends
                        ToppingDecorator
{
    BasePizza basePizza;
    public Extracheese(BasePizza pizza){
        this.basePizza = pizza;
    }
    @override
    public int cost(){
        return this.basePizza.cost()+10;
    }
}
```

is-a

```
public class Mushroom extends ToppingDecorator
{
    BasePizza basePizza;
    public mushroom(BasePizza pizza){
        this.basePizza = pizza;
    }
    @override
    public int cost(){
        return this.basePizza.cost()+15;
    }
}
```

→ Margherita + Extracheese.




```
BasePizza Pizza = new Extracheese(new Margherita());
```

2nd object creation.

1st object creation

→ Pizza.cost ??

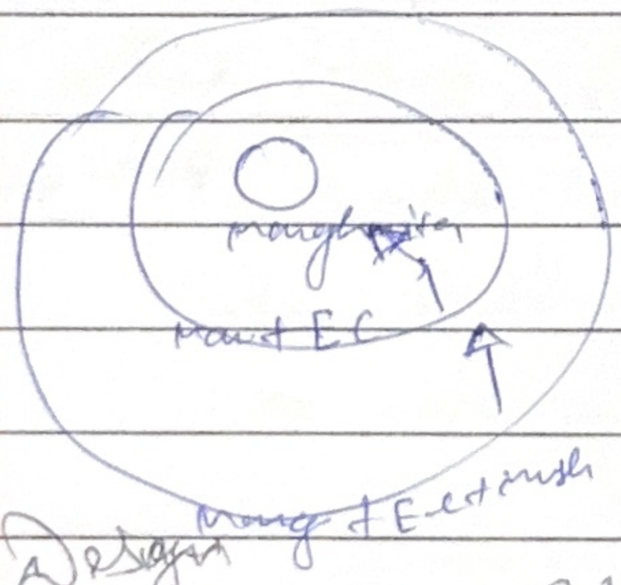
100 + 10 = 110

→ Mushroom + Margherita + Extra cheese

```
BasePizza Pizza = new mushroom(new Extracheese(new margherita()));
```

Pizza.cost = 15 + 10 + 100

125



* Design Coffee Machine

(V) Fascade Design Pattern

Date: / /
Page No.:

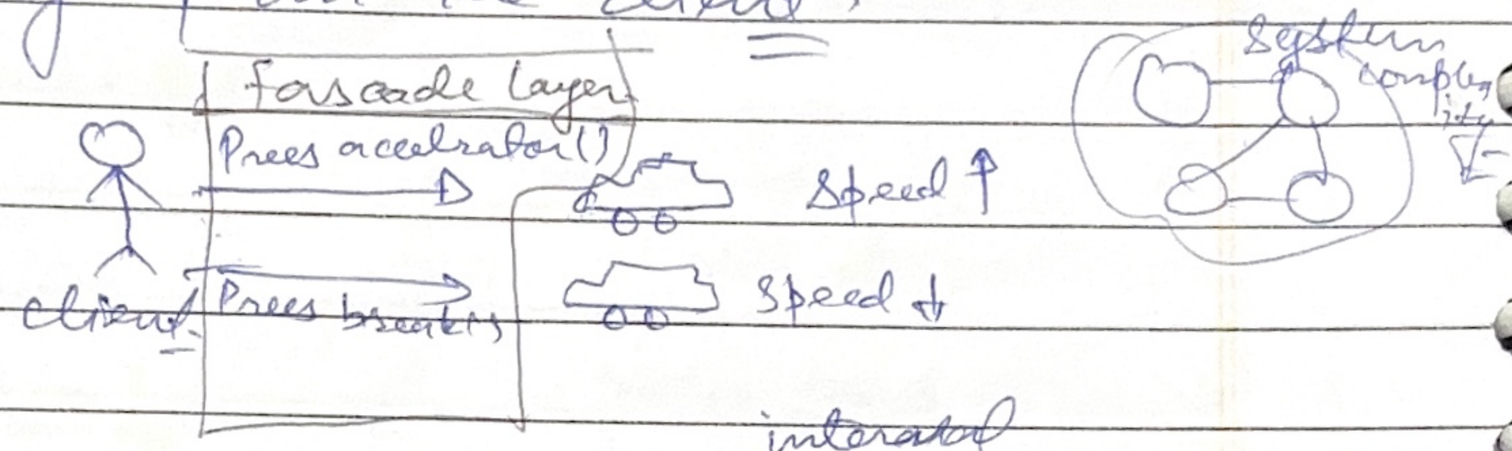
→ It is a structural DP that provides a simplified interface to a library, a framework, or any other complex set of classes.

→ It is simple and widely used.

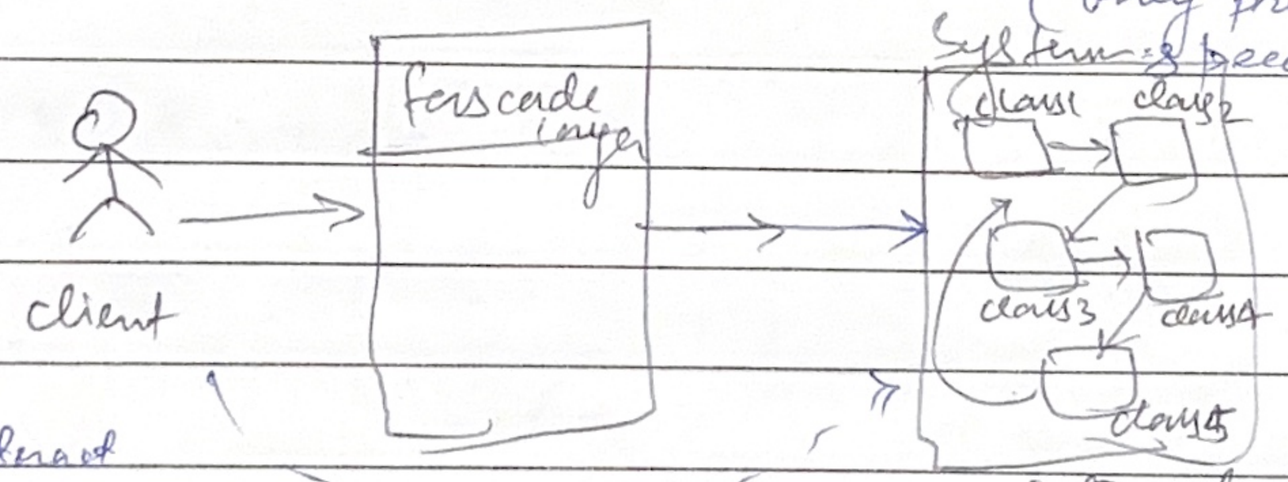
→ Real world use case
→ fascade vs Proxy
→ fascade vs Adapter

→ When to use fascade DP & why?
↳ When we have to hide the system complexity from the client.

→ Con:



→ client does not care about the system functional & interaction
only cares about the functionality
{ how speed goes up does not matter only press accelerator system speeds up }



→ client interact with fascade layer.
& fascade layer interact with internal system

→ We are not forcing client to always use fascade layer
they can also might use class 5, 2, 1 etc. or system.

→ client has always the choice.

DAO → Data access object.
A class which interact with database. (insert select delete update)

→ Fascade takes the responsibility to create the class using constructor of all the object it interacts with and it expose those method which only client needs it.

Date: / /
Page No.:

Senario 1 :- → public class EmployeeDAO{
    public void insert (){
        // insert into Employee Table
    }
    public void updateEmployeeName(){
        // updating employee Name
    }
    public Employee getEmployeeDetails (String EmpId){
        //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();
    }
}

"has-a"

→ public class EmployeeFascade{
    EmployeeDAO employeeDAO;
    public EmployeeFascade(){
        employeeDAO = new EmployeeDAO();
    }
    public void insert (){
        employeeDAO.insert ();
    }
    public Employee getEmployeeDetails(int empID){
        return employeeDAO.getEmployeeDetails(empID);
    }
}

"has-a"

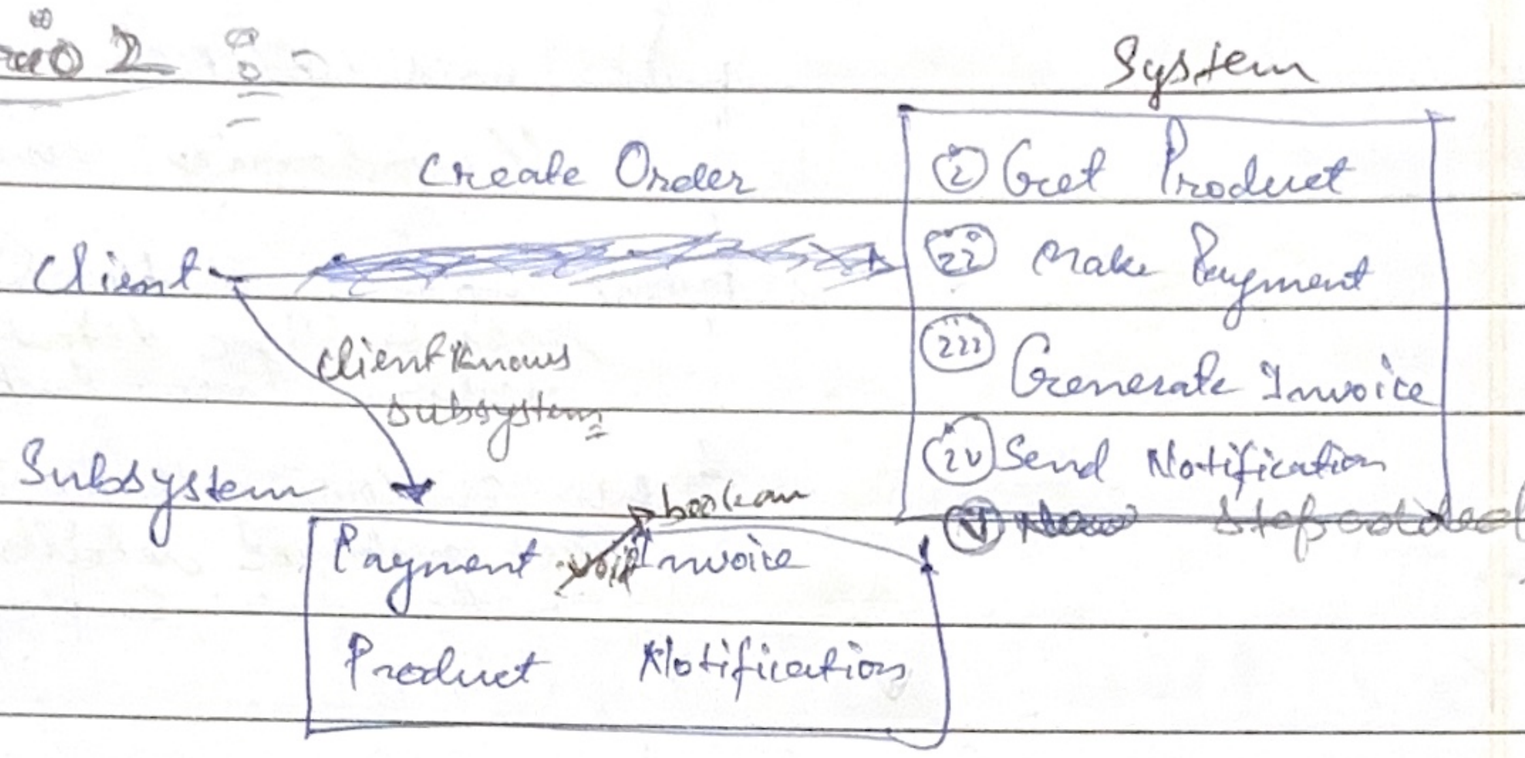
→ public class EmployeeClient {
    public void getEmployeeDetails (){
        EmployeeFascade employeeFascade = new EmployeeFascade();
        Employee employeeDetails = employeeFascade.getEmployeeDetails(121222);
    }
}

Here client only want employee Details by empId &
So, Fascade layer only do insert emp
No other methods are implemented.

Date: / /
Page No.:

* We use Fascade we have to expose only certain methods to the client.

Scenario 2 :-



Disadvantage :-

(i) In future if order creation logic changes ~~begin~~ adding a new step.

(ii) Any change in logic of classes that also impact client.
return type

Better way is to introduce Fascade layer so all the logic that client is keeping now takes care by fascade layer.

```
public class ProductDAO {
    public product getProduct (int productid) {
        return new Product 1); // get product based on product id.
    }
}
```

```
public class Payment {
    public boolean makePayment() {
        return true; //initiate payment
    }
}
```

```
public class Invoice {
    public void generateInvoice() {
        // this will generate invoices
    }
}
```

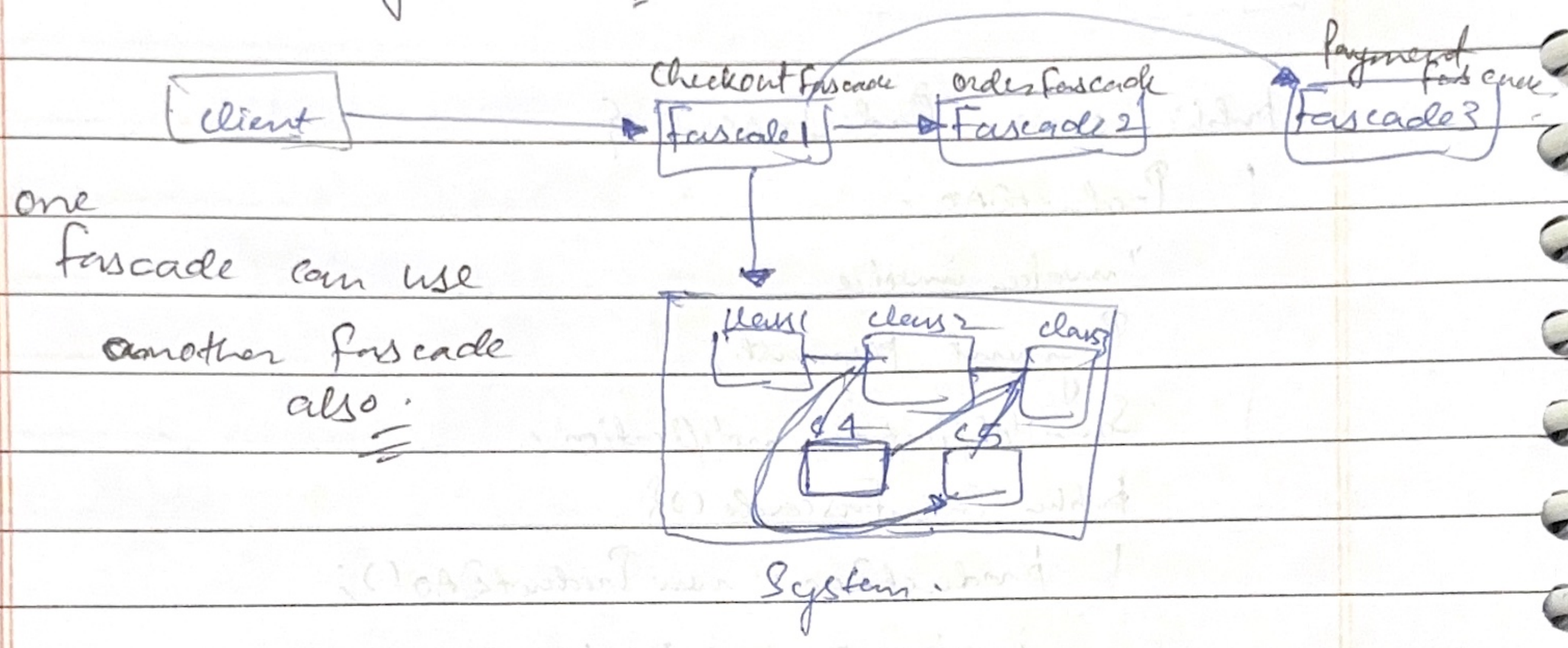
Date : / /
Page No. :

```
public class SendNotification {
    public void sendNotification () {
        //this will send Notification
    }
}

public class OrderFascade {
    ProductDAO ;
    Invoice invoice ;
    Payment payment ;
    SendNotification notification ;
    public OrderFascade () {
        productDao = new ProductDAO ();
        invoice = new Invoice ();
        payment = new Payment ();
        notification = new SendNotification ();
    }
    public void createOrder () {
        (1) Product product = productDao.getProduct (121) ;
        (2) Payment.makePayment ();
        (3) invoice.generateInvoice ();
        (4) notification.sendNotification ();
            //order creation Successful :
    }
}

→ public class OrderClient {
    public static void main (String args[]) {
        OrderFascade OrderFascade = new OrderFascade ();
        orderFascade.CreateOrder ();
    }
}
```

Date: / /
Page No.:

Senario 3:- Fascade using other fascade:

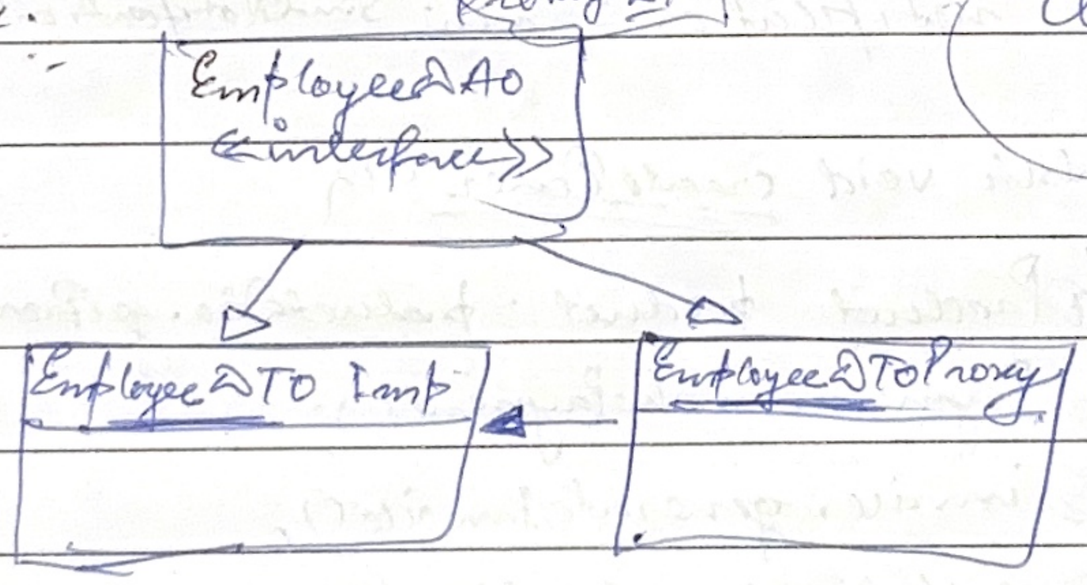
→ It can leverage the different fascade available which makes fascade pattern much more Powerful and easy to use.



one fascade can use another fascade also.

* Fascade Vs Proxy:-

Proxy:



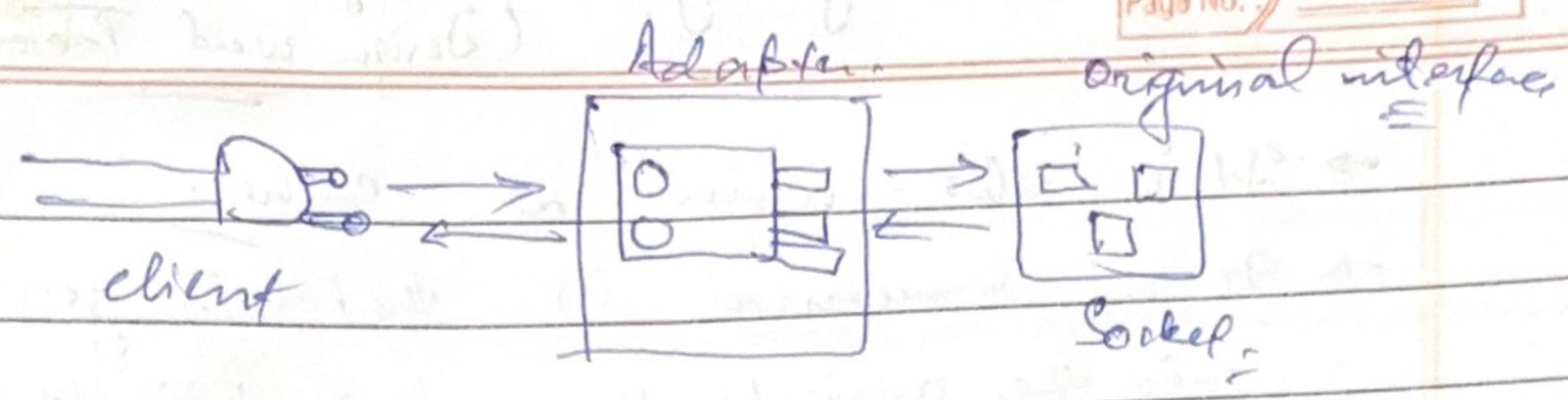
Proxy:
Client → Proxy → Actual Object

Fascade:
Client → fascade → Object. Similar

① Proxy is for a particular object (EmployeeDTO) implements same interface so Proxy is for the same interface (EmployeeDTO Proxy). Proxy takes care of only one object. (same object.

* Fascade Vs Adapter.

Client & original interface is incompatible.
Adapter ~~brings~~ is used to bring the compatibility b/w client & original object.



* fascade is used to hide the complexity b/w client & original object.

(vi) Flyweight Design Pattern
(Design word Processor)

→ It is also known as cache.

→ It is Structural DP that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

→ Two important Interview Qns :-
- Design word Processor ? (Text Editor)
- Design Game ?

→ This pattern helps to reduce the memory usage by sharing data among multiple objects.

Let's first see the issue for gaming senario's.
we have to create an army of Robot. Game

Army type → Humanoid Robot, Robotic Dog.

```
public class Robot{
    int coordinateX;   // 4 Byte
    int coordinateY;   // 4 Byte
    String type;       // 50 char = 50 Byte
    Sprites body;      // small 2d bitmap (graphic element) // 30KB
    Robot(int x, int y, String type, Sprites body){
        this.coordinateX = x;
        this.coordinateY = y;
        this.type = type;
        this.body = body;
    }
    // getter & setters
}
```

Extrinsic Data

Shared Data b/w 5 Lakh objects

Expensive

Intrinsic Data

```
public class Sprites{
}
```

// Body or graphics of Robot.

Army of

we want to create 5 lakhs of Humanoid Robots & and 5 lakhs of robotic dogs.

Date: / /
Page No.:

→ public class Main {
public static void main (String [] args){
int x=0; //
int y=0;

5 lakhs
for(int i=0; i<500000; i++){
Sprites humanoidSprite = new Sprites();
Robot humanoidRobotObject = new Robot(x+i, y+i, "HUMANOID", humanoidSprite);
}

&

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

Total 10 lakhs Objects are created and each objects takes some memory ≈ 30KB + 4B + 4B + 50B
So, ~~To~~ One object takes around ≈ 31 KB
Total space used by 10 lakh robot. ≈ 31KB × 10,00000
But, suppose interviewer gives only 25GB ≈ 37 GB in memory
then it will crashed.
or out of memory

* So, we have to come up with the solution which will optimize memory usage.

* Observe and understand, when to use this pattern:-
① when memory is limited.
② 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.
③ Creation of object is expensive.

Date: / /
Page No.:

This is how it solves the issue :-

(i) From object, remove all the Extrinsic data (Remove coordinate x, coordinate y) and keep Intrinsic Data (this object called Flyweight Object)

(ii) This Flyweight class can be immutable (private variable and only getter no setter)

(iii) Extrinsic Data can be passed to the Flyweight class in method parameter.

(iv) Once the Flyweight object is created, it is cached and reused whenever required.

```
public class interface IRobot {
    public void display(int x, int y);
}
```

Flyweight class

```
public class HumanoidRobot implements IRobot {
    private String type;
    private Sprites body;
    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.
    }
}
```

Intrinsic Data

flyweight class

```
public class RoboticDog implements IRobot {
    private String type;
    private Sprites body;
    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.
    }
}
```

* How to cache it?

Date: / /
Page No.:

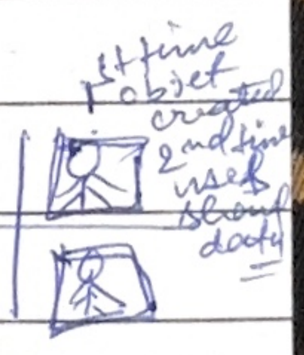
```
public class RoboticFactory {
    private static Map<String, IRobot> roboticObjectCache = new HashMap<>();
                                       object
                                       type human, dog
    public static IRobot createRobot(String robotType){
        if(roboticObjectCache.containsKey(roboticType)){
            return roboticObjectCache.get(roboticType);
        }
        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 roboticDogSprites = new Sprites();
                IRobot roboticDogObject = new RoboticDog(robotType, roboticDogSprite);
                roboticObjectCache.put(roboticType, roboticDogSprite);
                return roboticDogObject;
            }
            return null;
        }
    }
}
```

Before creating the object 1st it will check does this type exist in the cache if it will exist then it will return the same object.

If the object type is not created already or 1st time it will create the object type and stores it in the cache.

putting object type in the cache

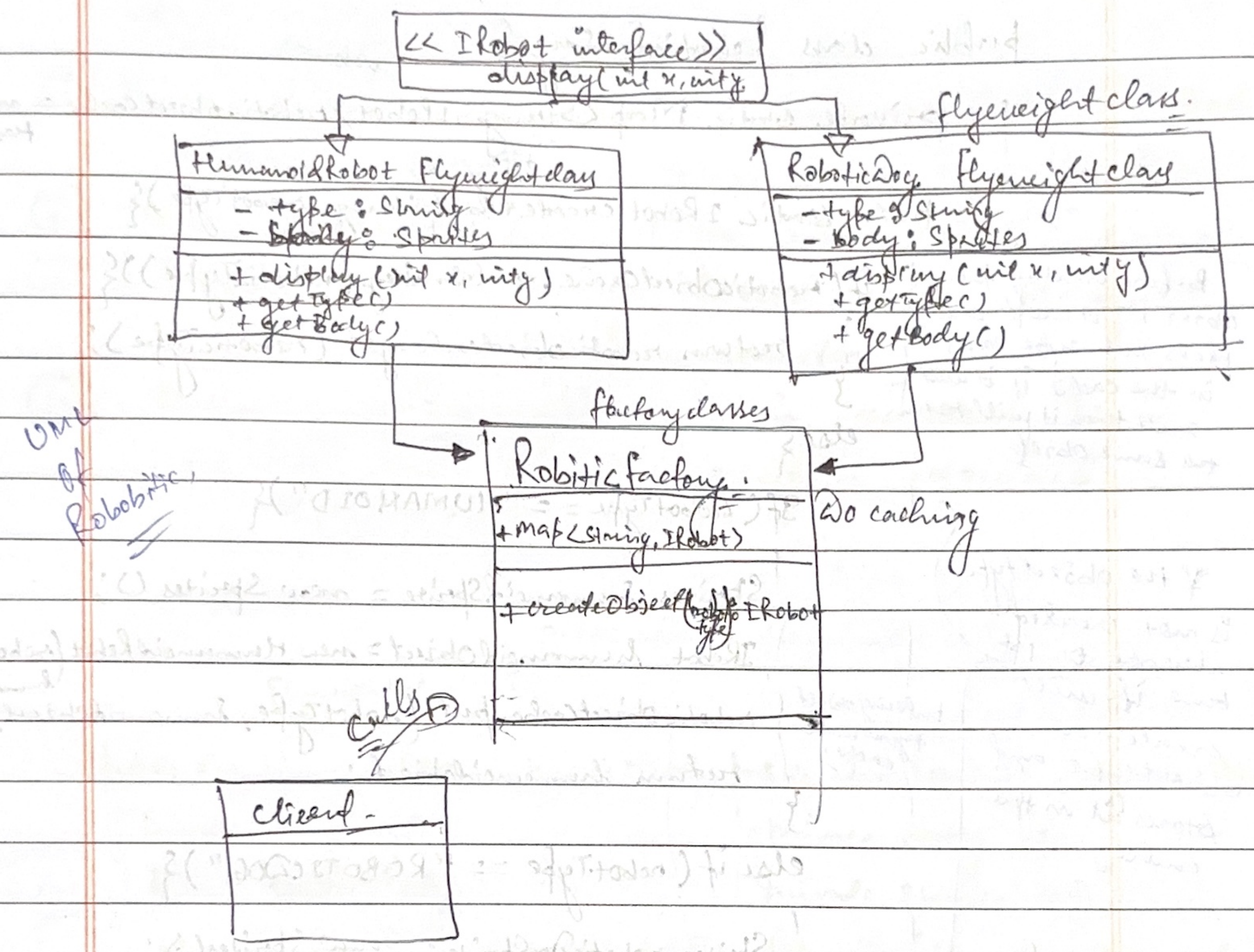
map = Humanoid, Robotic Dog



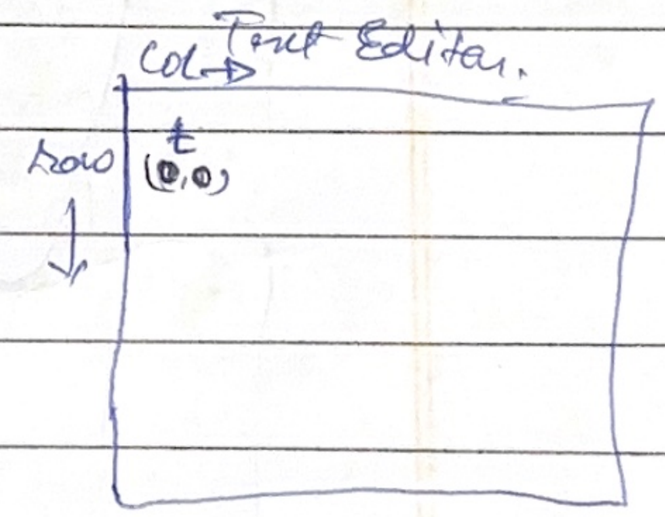

```
→ public class Main {
    public static void main(String args[]){
        IRobot humanoidRobot1 = RoboticFactory.createRobot("HUMANOID");
            humanoidRobot1.display(1,2);
        IRobot humanoidRobot2 = RoboticFactory.createRobot("HUMANOID");
            humanoidRobot2.display(10,30);

        IRobot roboDog1 = RoboticFactory.createRobot("ROBOTICDOG");
            roboDog1.display(2,9);
        IRobot roboDog2 = RoboticFactory.createRobot("ROBOTICDOG");
            roboDog2.display(11,19);
    }
}
```

Suppose 5 humanoid objects

5 robotic objects



* Example 2 :- Word Processor :-



Let's first see the issue of word Processor.

```
public class character {
    char character;
    String fontType;
    int size;
    int row;
    int col;
    character(char character, String fontType, int size, int row, int col) {
        this.character = character;
        this.fontType = fontType;
        this.size = size;
        this.row = row;
        this.col = col;
    }
    // getters and setters
}
```

// Intrisic Data

Extrinsic Data

Date: / /
Page No.:

/* this is the data we want to write into the word Prowd.
Total 58 character
t = 7 times
h = 3 times
a = 3 times and so on
*/

```
public class main {
    public static void main( String args [] ) {
        character obj1 = new character ( 't', "Arial", 10, 0, 0);
        character obj2 = new character ( 'h', "Arial", 10, 0, 1);
        character obj3 = new character ('i' ,"Arial" 10, 0, 2);
        character obj4 = new character ('s', "Arial", 10, 0, 3);
    }
}
```

(h = 3 times
t = 7 times)

* Here object are repeated and they shares some same variable size and fonts only vary the row & col. and this and labels of character in the editor.
so, you gonna creates labels of objects?
Solution is no, there will be memory issues in creating these huge objects.

Sol^n,

```
public interface ILetter {
    public void display( int row , int col);
}
```

```
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 col) {
            // display the character of particular font and size
                at given location.
        }
}
```

To make class immutable

Date : / /
Page No. :

```
→ public class LetterFactory {
    private static Map<Character, ILetter> characterCache = new HashMap<>();
                       Key                  value/object
    public static ILetter createLetter(char characterValue) {
        if(characterCache.containsKey(characterValue)){
            return characterCache.get(characterValue);
        }
        else{
            DocumentCharacter charterObj = new DocumentCharacter(charactervalue,
                                                                 "Arial",10);
            characterCache.put(charactervalue, characterObj);
            return characterObj;
        }
    }
```

2nd time from 2 check if the character object is already created in the cache if yes then return that

1st time If not created then create object and store in the cache

```
→ public class Main {
    public static void main (String[]args){
        ILetter obj1 = LetterFactory.createLetter('t');
            obj1.display(0,0);
        ILetterobj2 = LetterFactory.createLetter('t');
            obj2.display(0,6);
    }
}
```

1st time // new object letter is created and stored in the cache

2nd time // checks in the cache it is present then returned the already created object.

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

Total = 58 character

t = 7 time

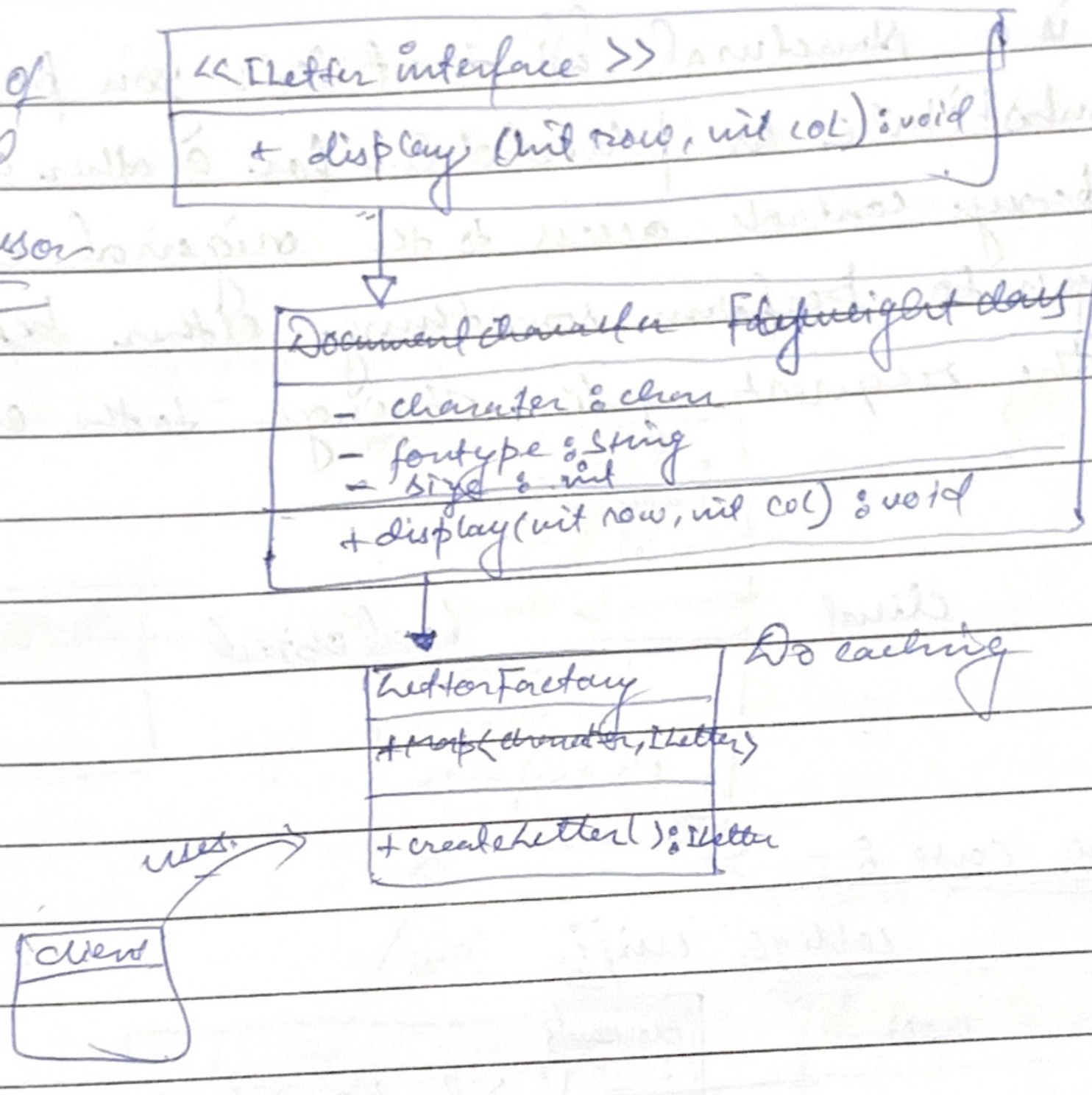
h = 3 time

a = 3 time

*/

Key	value
t	□
a	□
A	□
f	□

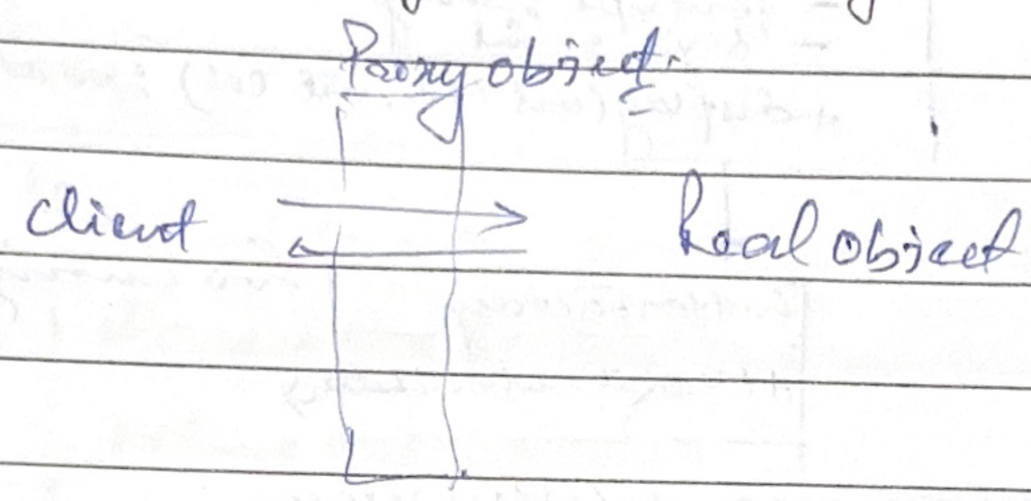
UML of Word Processor



(vii) Proxy Design Pattern

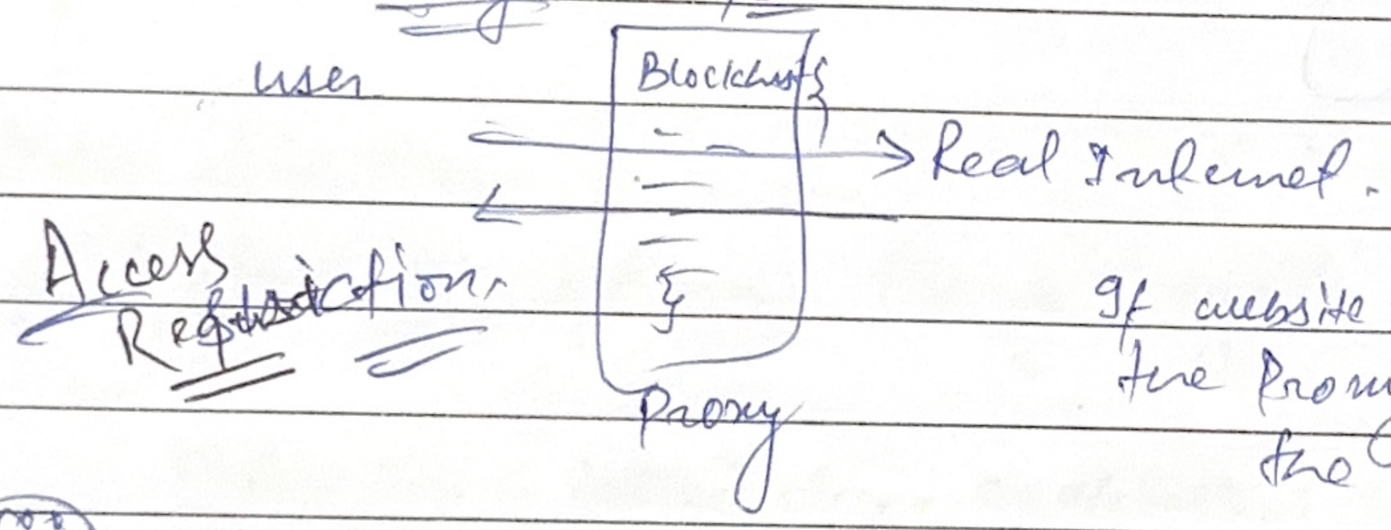
Date: / /
Page No.:

→ It is a structural DP that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object; allowing you to perform something either before or after the request gets through to the original object.



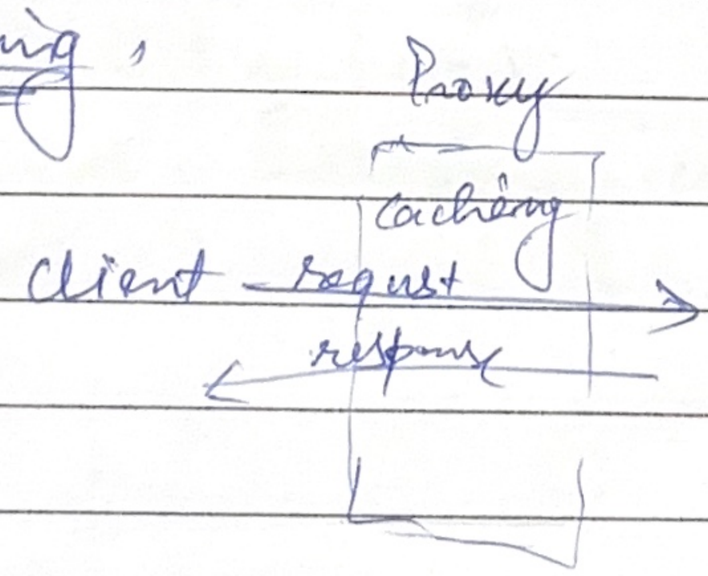
Use case :-

(2) College wifi :-



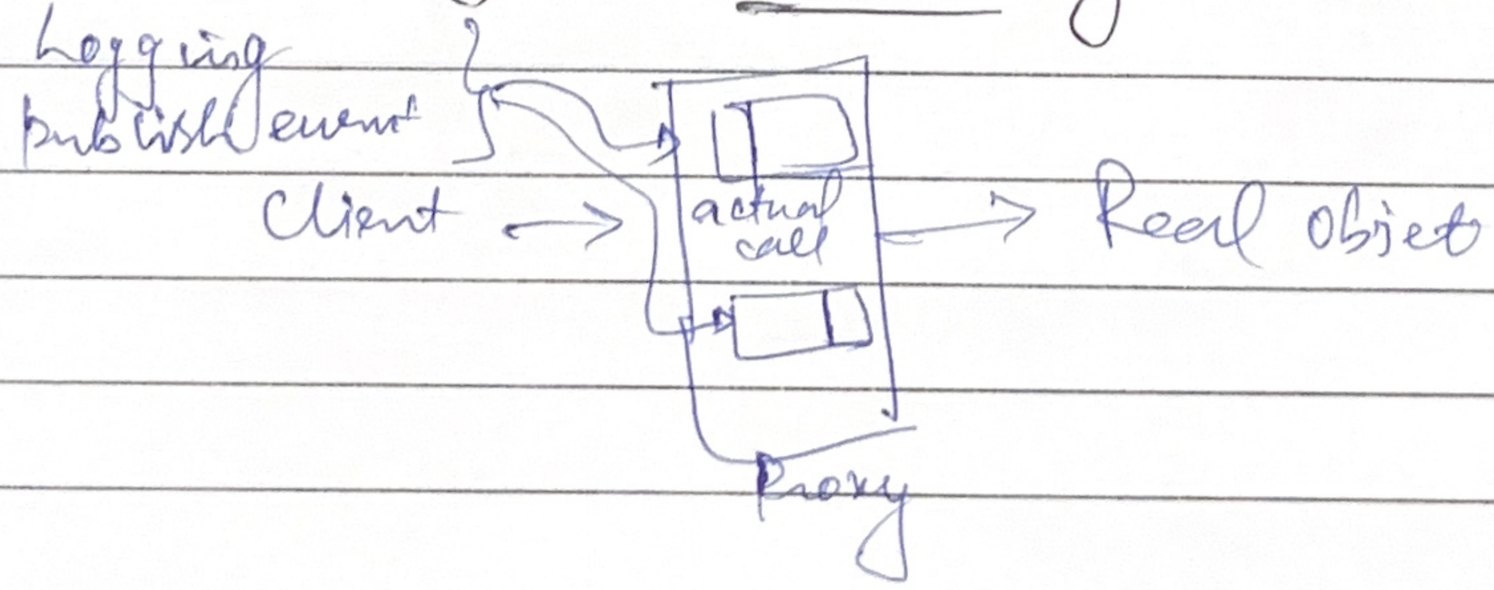
If website is in blocklist of the Proxy then Proxy rejects the request of the user.

(22) Caching :



If there is already a request stored in the cache then it will return from the cache itself not go the real object.

(222) Pre-Processing & Post Processing :



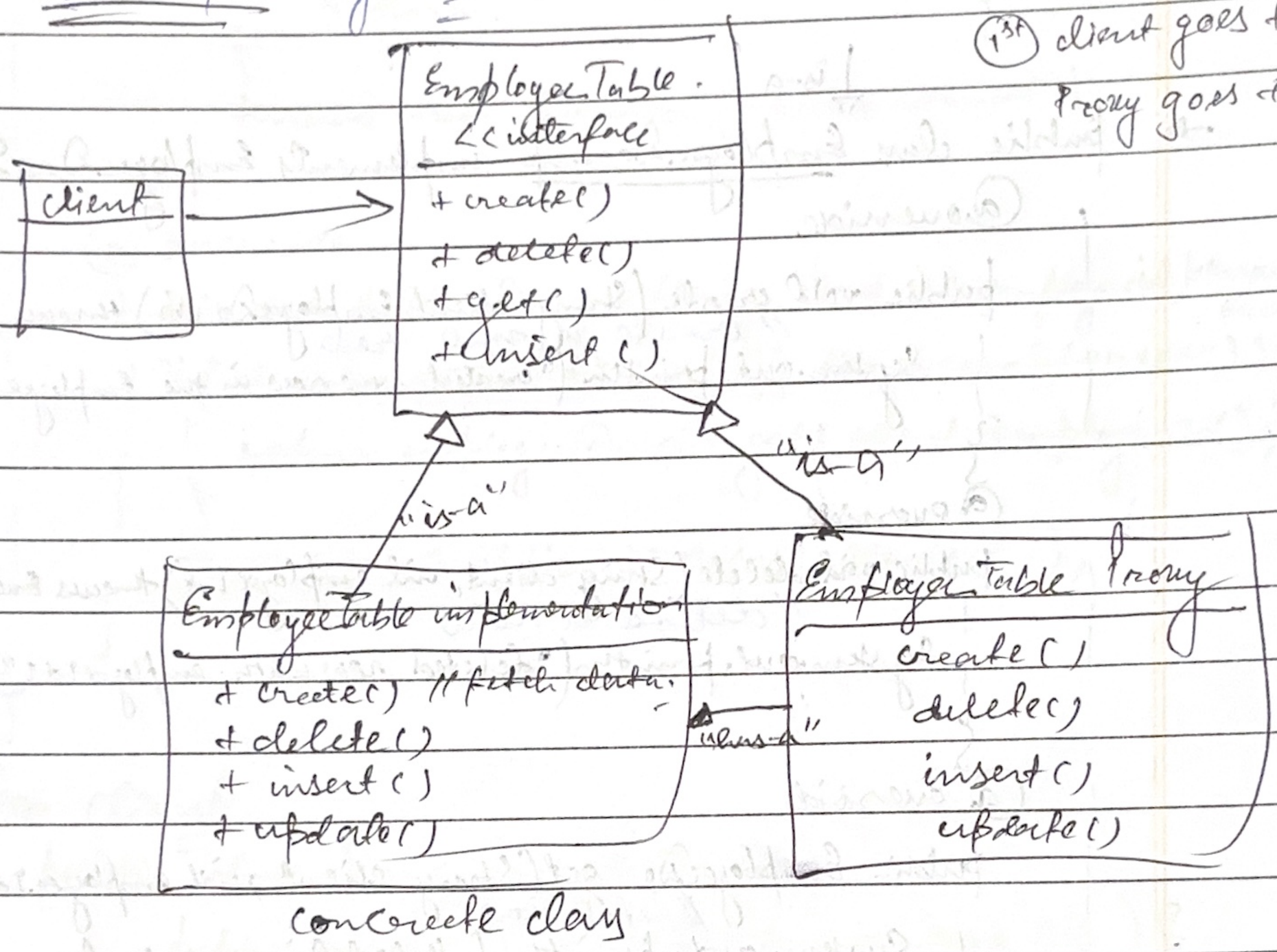
Date: / /
Page No.

(20) Springboot:

↳ Bean creation

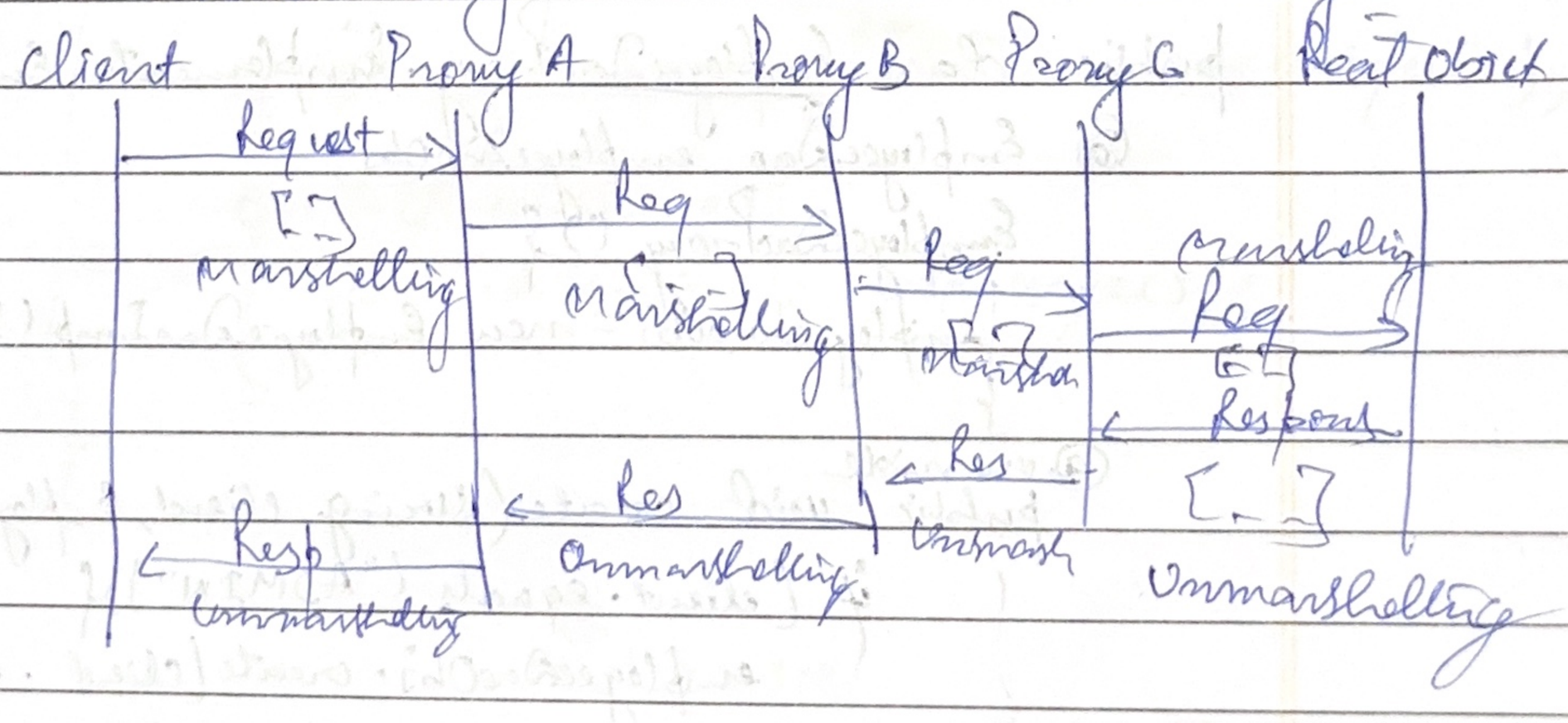
they uses (BeanProxy)

UML of Proxy DP:



(1st) client goes to Proxy
Proxy goes to table.

* There could be many Proxy in b/w client & Real object and they can use marshelling and unmarshelling.



Date : / /
Page No. :

→ 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;
}

↓ is-a

→ public class EmployeeDaoImp implements EmployeeDao {
@override
public void create (String client, EmployeeDo obj) throws Exception {
// Create a new row
System.out.println ("created new row in the Employee table");
}
@override
public void delete (String client, int employeeId) throws Exception {
// delete a row
System.out.println("deleted row with employeeId:" + employeeId);
}
@override
public EmployeeDo get (String client, int employeeId) throws Exception {
// fetch row
System.out.println("fetching data from the DB");
return new EmployeeDo ();
}
}

→ public class EmployeeDaoProxy implements EmployeeDao {
@ EmployeeDao employeeDaoObj;
EmployeeDaoProxy () {
employeeDaoObj = new EmployeeDaoImp ();
}
@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 EmployeeDao get(String client, int employeeId) throws Exception {
    if (client.equals("ADMIN") || client.equals("USER")) {
        return employeeDaoObj.get(client, employeeId);
    }
    throw new Exception("Access Denied");
}
}
```

→ Proxy client.

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

→ ① User.
↳ Access Denied

→ ② ADMIN
↳ created a new row in the employee table
Operation successful.

Date: / /
Page No.:

QNs

(i) Parking Lot
(ii) Design Tic Tac Toe game ✓
(iii) Elevator System ✓
(iv) Car Rental System ✓
(v) Snake & Ladder ✓
(vi) HashMap internal implementation ✓
(vii) BookMy show Design Movie Ticket Booking && concurrency handling ✓
(viii) Design Vending Machine ✓
(9) ATM Low level Design ✓
(10) Design chess game (Soumajit)
(11) LLD of splitwise ✓
(12) Splitwise simplify Debt / Optimal Accounting Balancing
(13) LLD of cricbuzz ✓
(14) LLD of inventory Managment system.
(15) Apply coupon on shopping cart
Design True caller
Design Car booking service like Ola, Uber
Design Online Hotel booking System (Soumajit)
Design Library Managment System - (Soumajit)
Design Traffic Light System
Design Meeting Scheduler
Design Online voting System
Design cache ~~management~~ mechanisms
Design Linkdin
Design Amazon (Soumajit)
Design Airline Managment System
Design Stock Exchange System
Design Learning Managment System
Design a calender Application.
Design (LLD) of Payment System

Date: / /
Page No.:

Design (LLD) of chat Based system.
Design food delivery app like Swiggy & Zomato.
Design community Discussion Platform
Design Restaurant managment System.
Design Bowling Alley Machine
Design (LLD) Rate Limiter.
Design Google Pay
Redis Internal.
Design Stack overflow (Sourayasit)