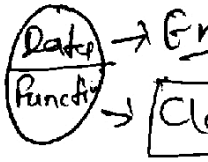
29/05/22. System Design.

Low level Design & High level Design.
[L.L.D] [H.L.D.]

Low level Design :-

Object Oriented Programming :-

→ Real world communication.




→ A way of programing that helps us to write code maintainable.
- easy to understand.
- easy to change.
- Not susceptible of errors.

=> Class → Blue print.
=> Objects → Objected blue print.
=> Inheritance
=> Polymorphism
=> Abstraction.
=> Encapsulation.

Class :- Class is a blueprint or structure or type.

Actual implementation is object.

Access Modifiers :- Restriction providing.
→ public, private, protected, default

→ Default can be accessed only within package.

Constructor: Creates Object for class.
- → It can be public, private, default.
- → No return type.
- → A default constructor is created automatically.
- → If we write an parameter const then default will be deleted.
- → We can write multiple custom constructors

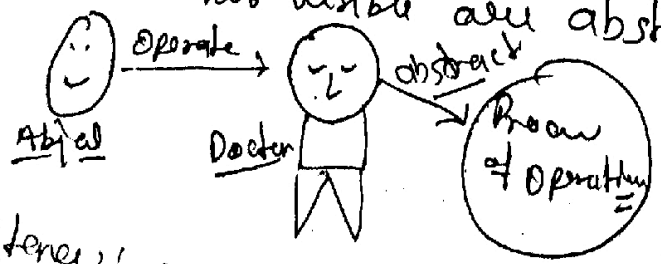
Static
The value which are common all the Objects are known as static value.

Final ⇒ Initialize only one.

this ⇒ refer to current Object.

Encapsulation: Binding the member and function which are related is known as encapsulation.

Abstraction: The between which are not public or not visible are abstraction for the user




Inheritance: Accessing the properties of parent class into sub class is known as inheritance.

Super: Refer to parent class Object.

Qwen Design Library management.
Design Hotel management.
Design Chess game → Bit manipulation are faster in calculation

30/5/23 S.O.L.I.D Principles

[S] Single Responsibility: Every class should have a single responsibility. Or should contain related data member, function.

```
Class Book
{ String name;
  public String getName() { return name; }
  public String printToConsole(text) { SOP(text); }
}
```

this method is not related, so it shouldn't be here.

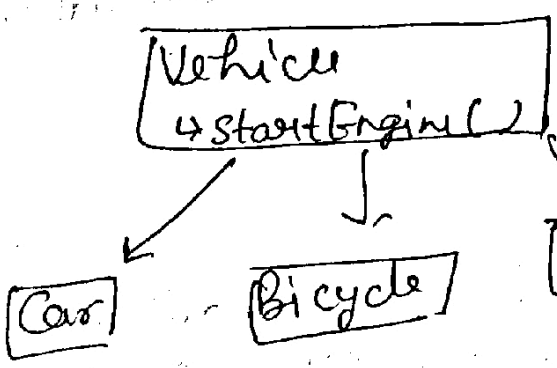
[O] Open for extention, closed for modification:

```
public interface Calculator
{ public double calculate(int a, int b); }

class Add implements Calculator
{ @Override
  Public double Caclulate(int a, int b)
  { return a+b; }

class Multiply implements Calculator
{ @Over
  pub class Calculate(int a, int b)
  { return a*b; }
```

## [L] Liskov Substitution! :- (Listov)

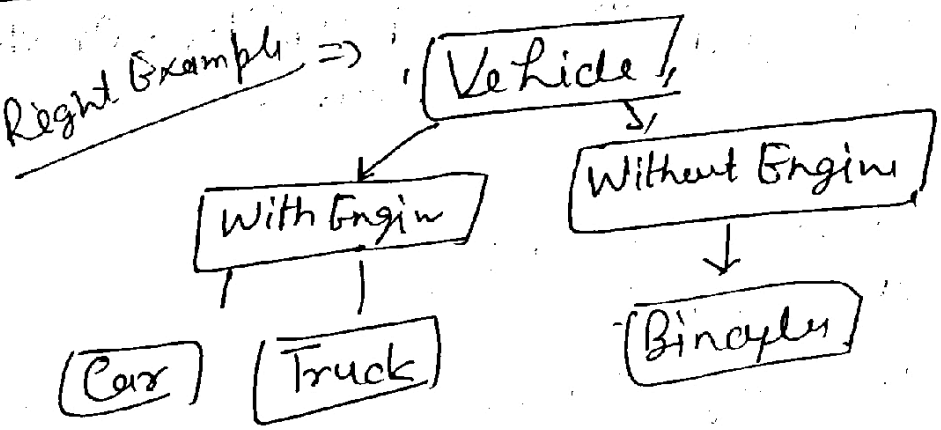
Derived class should be able to substitute the derived class.

```
BaseClass ben = new DerivedClass();
```

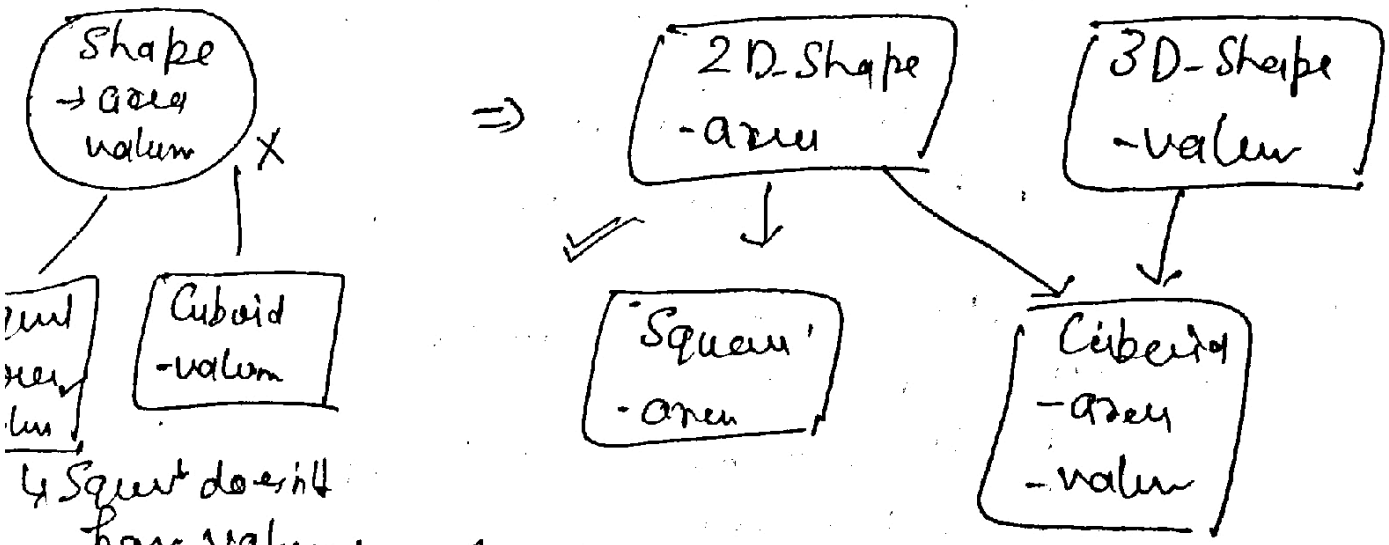



⇒ So here start engine method is accible in Bicycle class also, but Bicycle doesn't have engine so this is voilating Listov ≠

Right Example ⇒




## [I] Interface Segregation :-




↳ Square doesn't have value ≠

Interface should be segregated.

⇒ We shouldn't implement an interface for which we don't need some of the functionality.

[D] Dependency Inversion :- we should depend on abstraction.

⇒ High level module shouldn't depend on low level detail.

```
Class Desktop
{ private Monitor m;
  private QwertyKeyboard kb;
}
```

Now in future what if we want to change to a gaming keyboard.

⇒

```
Class Desktop
{ private Monitor m;
  private Keyboard kb;
  public Desktop(Keyboard kb)
  { this.kb = kb;
  }
}
```

This is fine as Desktop is not depend on keyboard

30/5/22 ☆

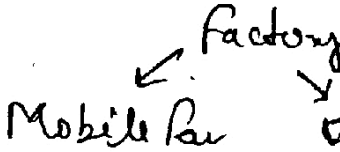
## Design Patterns :- General tips & tricks.

→ Factory Design Pattern :- There should be a single factory to create the instances.

```
Interface OS
{ printNow;
}
Class Window implements OS
{ printNow
  { sop("Window");
  }
}
```

```
Class Factory implements OS
{ public OS getInstance(String name)
  { if(name == "Window")
    { return new Window();
    }
  }
}
```

Abstract Factory :- Factory of factories are known as abstract factory.



## Singleton Design Pattern :-

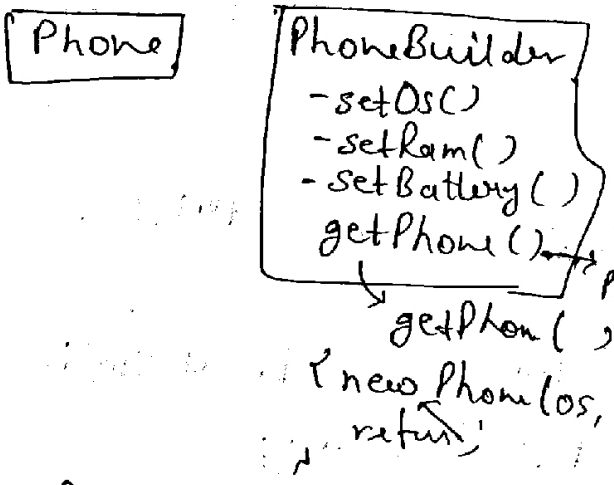
Steps:
1- Static object
2- Private constructor
3- getInstance.

Useful when we need single object throughout.
ex- Logger
Exception handling.
Reading Object.
DB Context

31/8/22.

## Builder Design Pattern :-

We should have setter function for properties. And an method to get the actual object.




Declaring the constructor for each new property is not a good idea.

## Prototype Design Pattern :-

If we are creating the multiple object of same class, then we should create deep copy.

```
class Bookshop
{
  public Bookshop clone()
  {
    Bookshop b = new Bookshop();
    // Load all property values of this object
    return b;
  }
```

If we will use default clone() method then it will create shallow copy.

Adapter Design Pattern!- Adapting the things from one form to another form is represents Adapter Design Pattern.

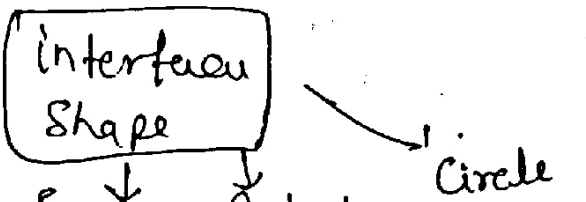
```
interface Pen
{ public write();
}
```

```
class PilotPen
{ public mark()
  { SOP("Hello written");
  }
}
```

```
class PenAdapter implem Pen
{ PilotPen pen = new PilotPen();
  @Override
  public write()
  { pen.mark();
  }
}
```

} Adapting the things.

Facade Design Pattern!- (Facade + main entrance)
Reducing the inner complexity by a facade is know as Facade D.P.




```
ShapeMaker
  Square sq;
  Rectangle rec;
  Circle cir;
  drawSquare()
    sq.draw()
  drawRectangle()
    rec.draw();
  drawCircle()
    cir.draw();
```

Here ShapeMaker is facade.

Bridge Design Pattern!- Creating the bridge between two or more common implementation is know as the bridge design pattern.
=) Creating bridge between abstraction.

abstract class View
{ IResource res;
Public View(IRes r)
{ this.res = r;
}
public String show();
}

Bridge

interface IResource
{ String snippet();
String title();
String image();
String url();
}

class ArtistRes : IRes
{ implementation of these 4 methods
}

class LongFor : View
{ public String show()
{ this.res.snippet();
this.res.title();
return html;
}

Structural Design Patterns.

## Composite Design Pattern :- (Structural Design Pattern)

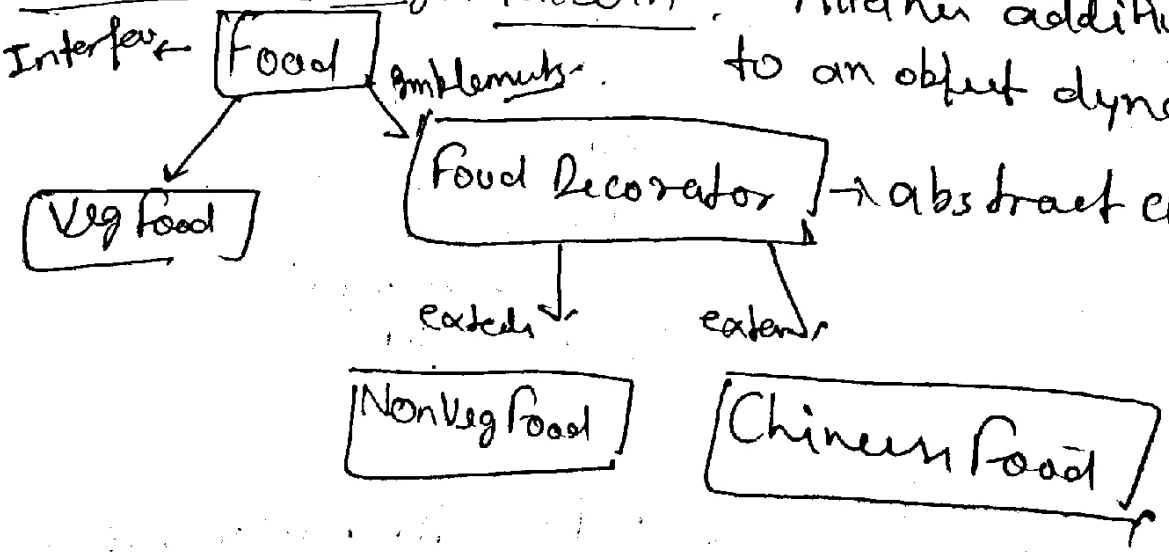
interface PriceCalculate

class Product.

class Box
{ products.

We use when we have tree type structure and have common functionality.

## Decorator Design Pattern :- Attaches additional responsibility to an object dynamically.



Design ATM Machine:-

We have coin { 20, 50, 100, 200, 500 }
give center fu each → deposit( )
withdraw( )

Design Simple Bank System

operation → transfer (from, to, money)
withdraw (account, money)
deposit (account, money)

Design Parking Lot:- (Small)

big, medium, small.
↓ ↓ ↓
type:= 1 2 3

public boolean addCar(int cartype)
{ if(data[cartype-1] > 0)
{ data[cartype-1] --; return true;
}
return false;

Enums:- When we have fixed numbs of valw then we create enum.

enum Color
{ RED, GREEN, BLUE; }

=) we an add constructor also

Abstract Class:- Don't have Objects.

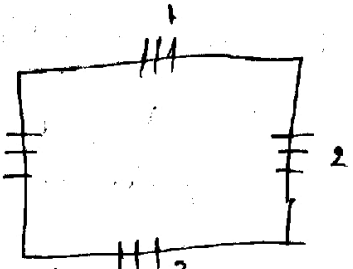
abstract class Animal
{ age, nam.
public method
}

Cat extend Animal
Dog extends Animal
Horn extend Animal

# Design Parking Lot: (Big)

Requirement.

1- Big parking (10k - 30k).
2- 4 entrances - 4 exits.
3- Ticket & spot assigned at entrance.
4- Parking spot should be nearest to entrance.
5- Limit / Capacity - 30k.
6- Parking spots - Handicaped, Compact, Large, moto.
7- Hourly rate.
8- Cash & Credit cards.
9- Monitoring system.



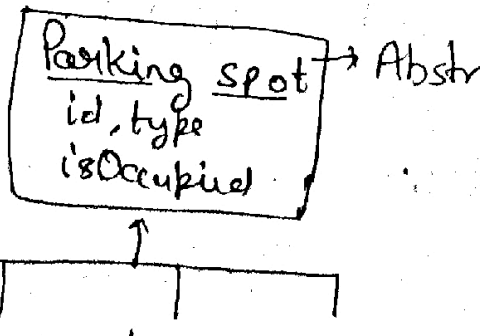

Mistakes

=> Superficial
=> Vehicle class is not required.
=> Show available spots

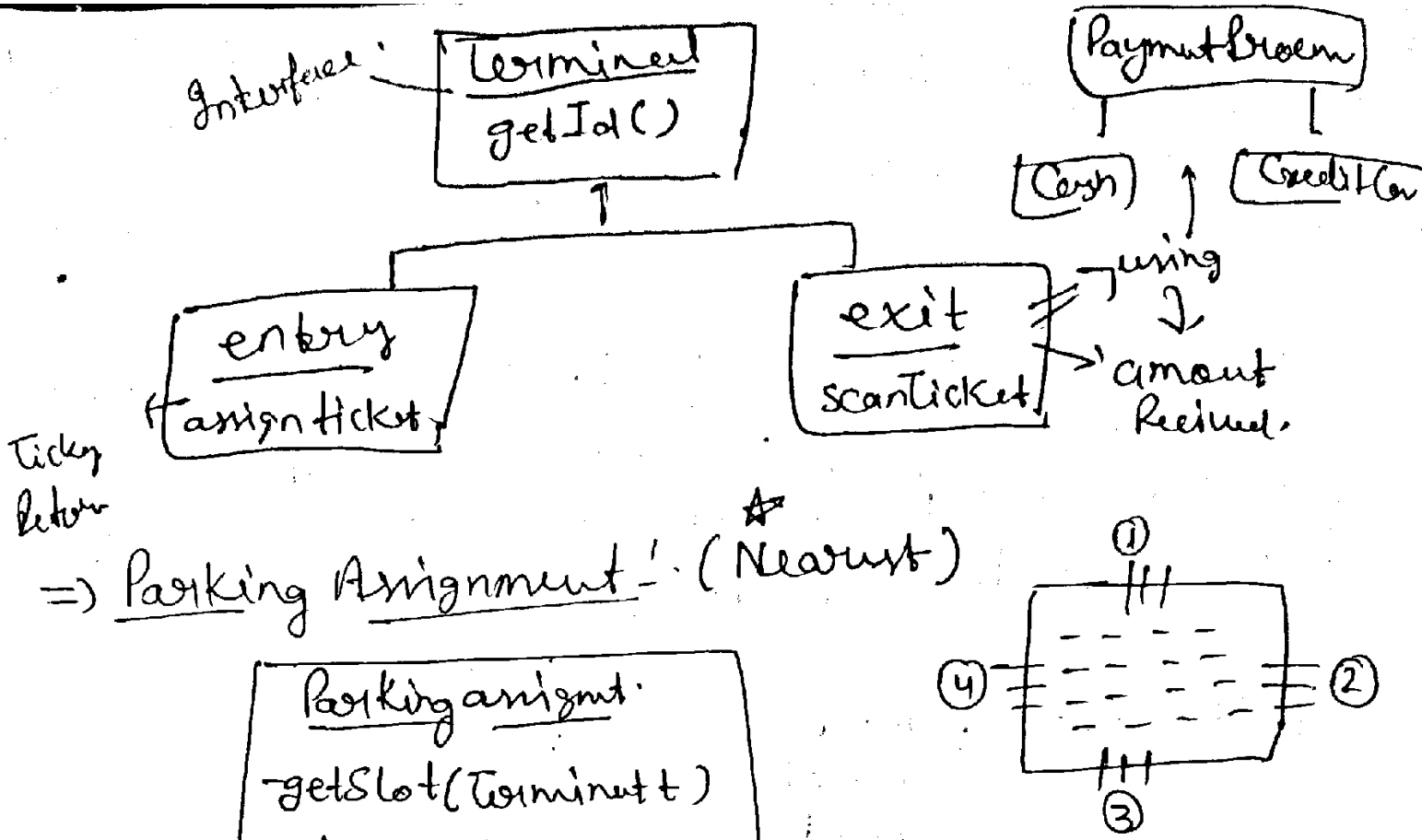
Steps:

=> Actors.
- Parking lot
- Entry / exit (terminal)
  - print ticket
  - payment procem / scan ticket.
- Parking spot
- Ticket
- Database
- Monitory system.

=> Define Classes

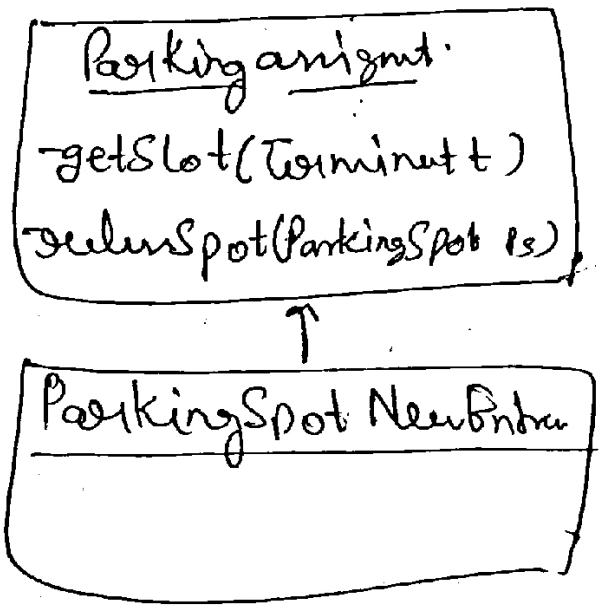



→ Abstract class.




=> Parking Assignment :- (Nearest) ☆



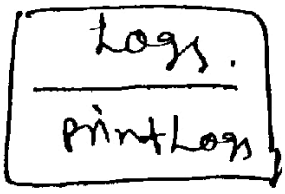




=> To get the minimum distance we will be using "Priority Queue" & Min-heap.

assign
Release.

=> DB . → No class, as this is storage thing.

=> Monitoring system :-



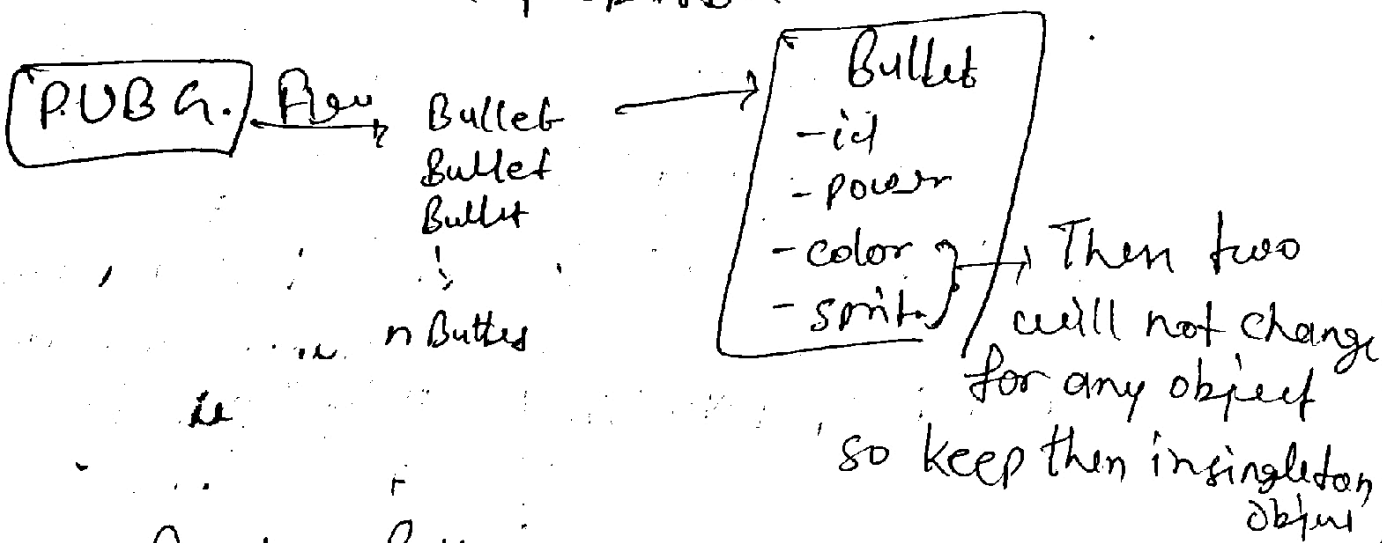

=> Singleton Design Pattern Used :-

static ParkingLot pl :
getInstance()

## Flyweight and Proxy Design Pattern:-

It is a way to devide the properties in two parts extrinsic and intrinsic (immutable)
(mutable).

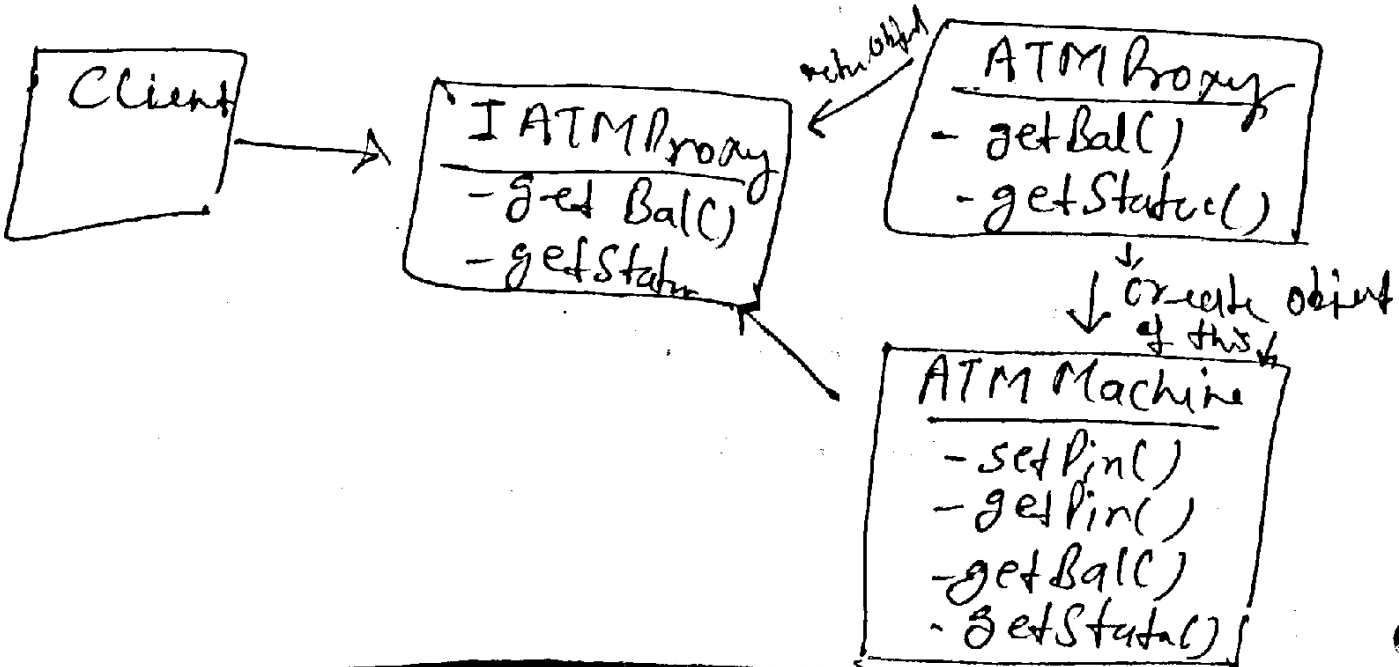
2) When create lot of objects and some property are not changing for any object then it is recommended to create an sigleton class for those immutable properties




## Proxy Design Pattern:-

This pattern use to add security to your design.

If we don't want to expose all the methods to the client, then we creat a Proxy of Actual class with only exposable data.



## Design Browser History:

history[] = new String[500];

current, max.

- easy hai bhai itna to kar.

* 06/06/21:

## Design Big Parking Lot :-

→ Multiple floor
- Multiple gate.
- An attendant at each gate.
- entry gate pr un ticket.
- Exit gate pr un payment on hourly basis.
- Should have parking facility of different types of with of vacant spots on the floor
- Display board at each floor showing the details

→ Firstly analyse bro model + bro → ParkingLot.

① class ParkingLot → Singleton.
{ List<ParkingFloor> pf;
List<Entrance> entrance;
List<Exit> exits
Address add; → isParkingAvailable(Vehicle)
String name; → updateAttendant(ParkingAttendant p)
}

② ParkingFloor:
floorID;
isFull
List<ParkingSpace> ps;
displayBoard

③ Gate - gateID
PA attendant
↓
④ Entrance exta Gate/ Assign Ticket
Exit exta Gate → pr un Payment

⑤ Address
- street
- city
- country.

SpaceID
IsBusy
CanBeParkedVehicleType

(7) ParkingDisplayBoard
Map<ParkingSpaceType, Integer> map
updateFreeSpotAvailable(Type, space)

(8)
Account
name
email
mobile
empId
address

abstract
Admin extends Account
addParkingFloor()
addSpace()
addBoard()

Parking Attendant extends Account
Payment paymentService
vehicleEntry()
processPayment()

(Any)
enum PaymentType
- Card, Cash, UPI

enum ParkingType
- Car, Bike, Bus, Truck

(9)
Vehicle
LicenseNumber
type
ticket
paymentInfo

(10)
Ticket
id
VehicleId
spaceID
entryTime, exitTime
spaceType
cost
ticketStatus
updateTotalCost()
updateVehicleExitTime()

(11)
PaymentType
- VehicleType
- Bus, Bike, Car

Payment
- makePayment()

PaymentInfo
- amount
date
txid
ticketStatus

enum ParkingTicketStatus
- PAID, ACTIVE

enum PaymentStatus
- Unpaid, Pending, Completed, declined, Cancelled, Refunded

# Design ATM - LLD.

Requirnt:
- able to swap card
- withdraw funds from acnt of any bank
- deposit cash and chaqu
- transfer funds.
- check balance

Class ATM
- id;
- Address;
- CashDispenser cd;
- Screen screen;
- CardReader cr;
- Keypad keypad
- CashDeposit cdp;
- CashWithdraw cw;
- ChequeDeposit cqd;
- * BankService service;

Interface BankService
- isValidUser(User u);
- getCashDetail()
- executeTransaction()

CashDispenser
- Map<CashType, Integer> cash;
- dispenseCash()

Address
- city
- state
- ...

Cash
- type
- serialNumber

enum CashType
- ATM, Funds, Author

CardReader CardInfo
- CardType cp;
- Bank bank
- number
- cvv
- withLimit

CardReader
- CardInfo fetchCardDetail();

KeyPad
- getInput()

Screen
- display()

BankA implements BankService

BankB implements BankService

* BankServiceFactory
- getBankServiceObject(CardInfo, card);
↳ BankService return type.

↓ Factory design pattern.

Class Customer
- name
- AccNumber
- Status
- Account
- BankServ

Account
- number
- balance
accountType

enum CustStatus
- Blocked, Active
Closed.

Transaction
id
sourAccount
date

Deposit extends Transaction
- amount

CheqDep extends Transaction

extends

CashDeposit
- getCash()

extends

ChequeDeposit
- getCheque()

extends

WithDraw extends Tran
- withDrawAmt()
- amount

Transfer extends Transaction
- destAccount
- amount
- isValid()
- Status

TransactionDetail
- Status
sourAcNum
- date
type
id

enumTransactionStat
- pending, cancel, compl, error

enum TransactionType
- withdraw
- deposit
- transfer

# Design TicTacToeGame!

Requirmt's:
- n x n Board
- 2 player game.
- Undo a move / Undo multiple moves.
- check if a use made a valid move
- declare winner, - leaderBoard

Steps - Clear req.
- Planning structure
- Code

Basic - Board (Game)
- [][] board.
- initialize(): -1,
- getBoard()
* - getWinner();
- getCurrPlayer
* - makeMove()

Move
- Player
row
Clow

User
- userId
- stats -
- updateStat()
- gameplayed()

Stats (LeaderBo)
- wins
- loss
win%

Game
- Id
- user1
- use 2
- List<Move> moves.
- initialize(use1, use2);
- undo() → Removed last Mov list

LeaderBoard wins % sort!
- PriorityQue<Users>
- displayBoard()