Design Patterns

Class

It is a blueprint contains fields and methods

Instance Variables → Declared in class

Local variables → Declared in method

Encapsulation → Wrapping up of data under single unit. Ex- class

Abstraction → Hiding unnecessary details.

IS A vs Has A?

If IS A satisfies then former one is subclass, latter-superclass

Is a "Dog" an "Animal"? ✓

If Has A satisfies, former class has fields including latter one

"Dog" has a "height" ✓

Note → use static methods to reflect changes in parameters created in main functions' object

```
main()
{  Dog fido = new Dog();
   int randnum = 10;
   fido.changevar(randnum);
S.o.println(randnum) → 10 not 12 ✗
```

```
Dog →
{
public void changevar
        (int randnum
{
  randnum = 12;
}.
```

so to change →.

```
public static void changevar(int randnum)
{     randnum = 12;
}
```

Now → randnum = 12 ✓

Animal

setname()
setheight()
setweight()
setsound()

Dog extends Animal

digHole()
Dog(){
super();
setsound("Bark");

Cat extends Animal

Cat(){
super();
setsound("Meow");
attack();

WorkwithAnimals
main() → int justAnum = 10;

{
Animal doggy = new Dog();
Animal kitty = new Cat();
s.o.p(dog.getsound()); → Bark
s.o.p(kitty.getsound()); → Meow

doggy.digHole(); → This doesnot work.
Because we are referencing to animal type class.

Note

We cannot reference methods/field that are not in animal class.

* we need to type cast the object for using it →
((Dog)doggy).digHole(); ✓

* We cannot use non-static fields & methods in static method
s.o.p(justAnum); X → can't make static reference to non-static field justAnum

* We cannot use any private method defined in class.
To access this private method, we need public method.

Abstract Class

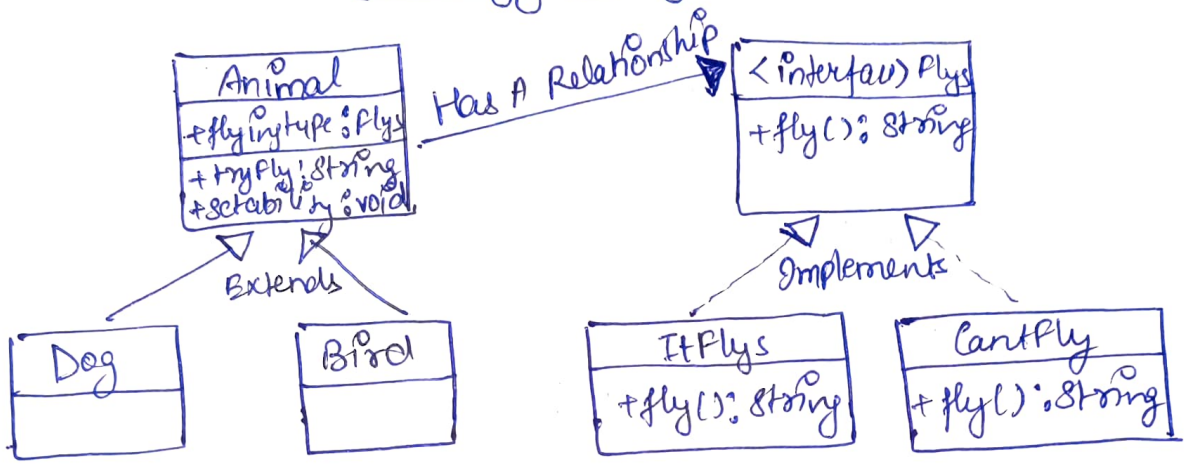
~~public~~ abstract public class creature.

* There are no abstract fields
* All methods don't have to be abstract
* You can have static methods.
* protected fields will be inherited and accessible by subclass.

Interface

* A class with only abstract methods
* You can add many interfaces to a class using implements
* You can only use public static & final fields
* We don't need to add abstract keyword because the methods are by-default abstract in interface

Strategy Design Pattern



Define a family of algorithms, encapsulate each one, make them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it.

When to use?

* When we want to define class that will have one behaviour that is similar to other behaviours in a list.
* When we need to use one of several behaviours dynamically
* often reduces long lists of conditionals
* keeps class changes from forcing other class changes
* can hide complicated / secret code from the user.
* Avoids duplicate code

CONS

Increased no. of objects / classes.

Creational Patterns

They are concerned with the way of creating objects.
Used when decision must be made when creating an object.

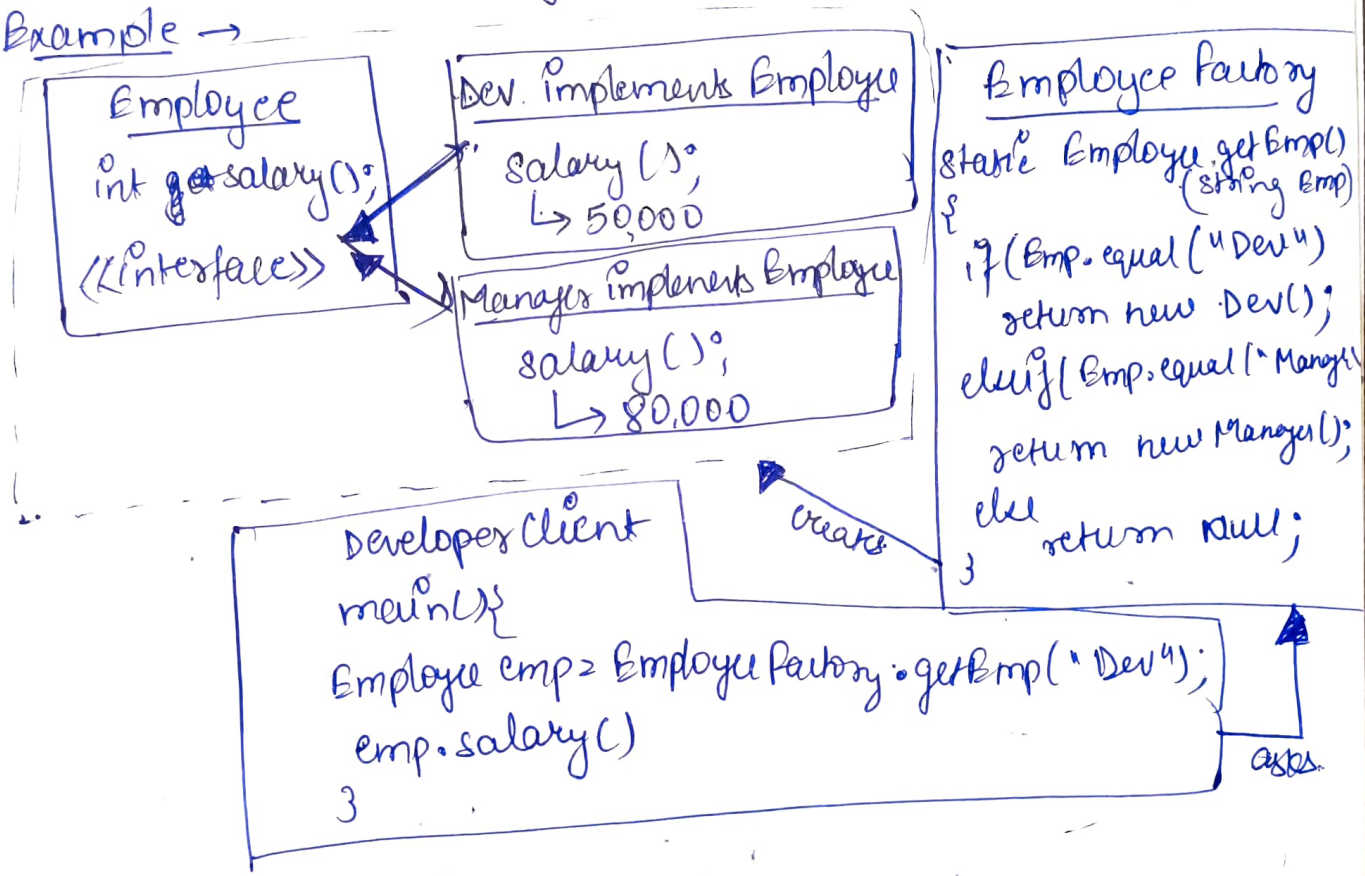
Factory Pattern → (Factory Method Pattern)

When there is superclass and multiple subclasses and we want to get object of subclasses based on input and requirement.

Advantage

* Allows sub classes to choose the type of objects to create.
* Promotes loose-coupling, more robust code

Example →

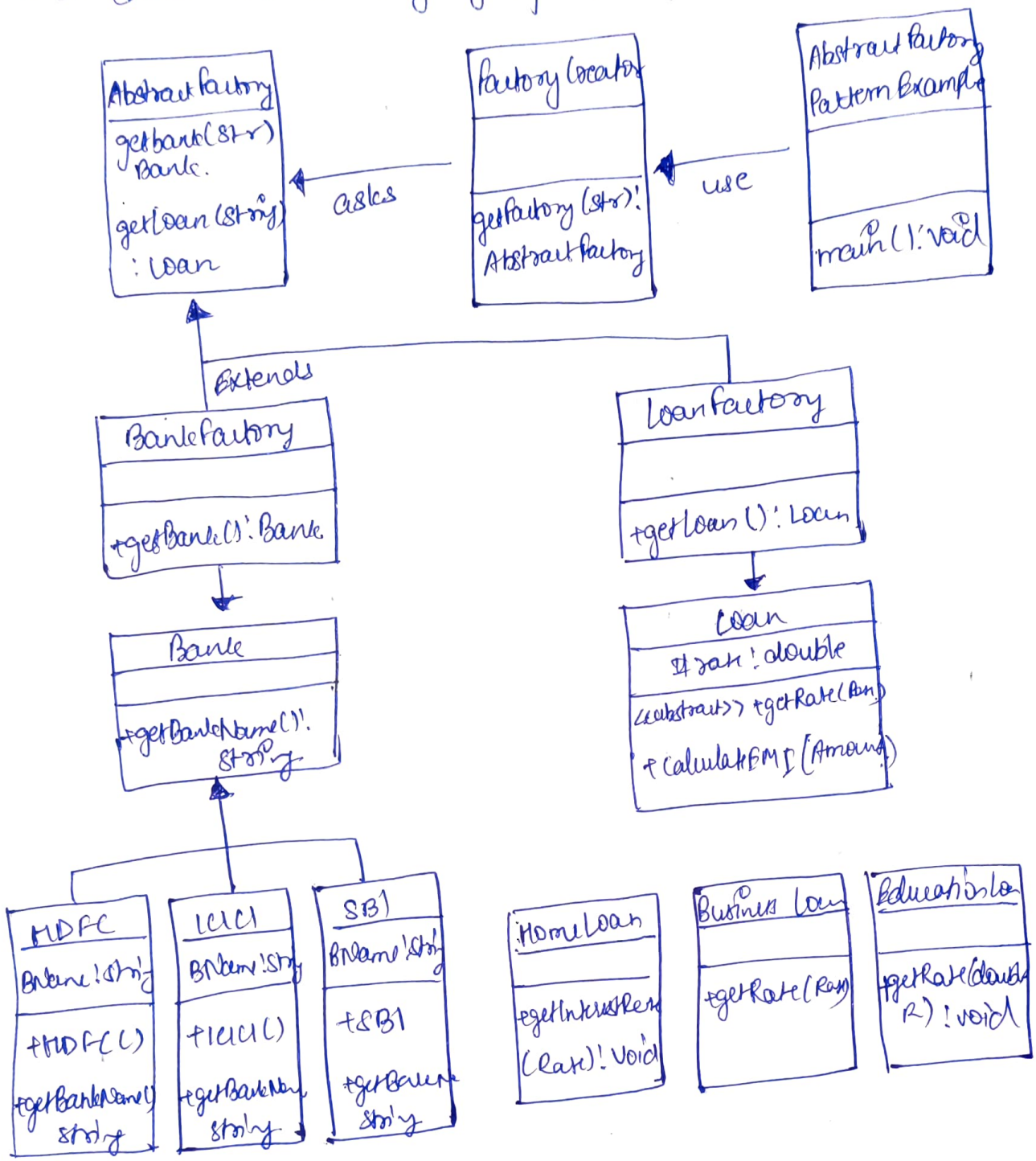


Abstract Factory Pattern →

Just define an interface or abstract class for creating families of related objects but without specifying their concrete classes. Abstract factory lets a class returns a factory of classes.

Advantage

* It isolates the client code from concrete classes.
* It eases the exchanging of object families.



Usage:

* When system needs to be independent of how its object are created, composed and represented.
* When family of related objects has to be used together, then this is used.

Singleton Design Pattern

Only one instance of the class should exist.
Other classes should be able to get instance of singleton class.
Used in logging, caching, session, Drivers

Implementation →

* Static private instance type
* Private constructor → To prevent multiple instantiation of class
* Static public method for returning instance

Initialisation Type

- Eager Initialisation →
- Lazy Initialisation
- Thread Safe Method Init^on
- Thread Safe Block Init^on

Eager Init^on

As soon as we initialise the instance, it automatically creates.

```
class singleton {
    private static singleton instance = new singleton();
                                  └→ object created while initialising.
    private singleton() {}
    public static singleton getInstance() {
        return instance();
    }
```

Lazy Initialisation →

Whenever there is need or call, then only it will create an object

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

Thread Safe Method Initialisation →

When two users are calling the class at same time, it may create 2 objects so it's not thread safe.
So we put them in synchronize, only one user can access at a time.
Here we are making whole method synchronise, both read and write access (object creation).

```
public static synchronised singleton getInstance() {
    if (object == null){
        instance = new singleton();
    }
    return instance;
}
                    // only one thread is accessing the whole method
```

Thread Safe Block Initialisation →

Rather than making whole method synchronise, we will make a block into synchronise. for write access.
For read access, there is no need to make it synchronise.

```
public static singleton getInstance() {
    if (instance == null) {
        synchronised (singleton.class) {
            if (instance == null){
                instance = new singleton();
            }
        }
    }
    return instance;
}
```

Prototype Design Pattern

when we want to avoid multiple object creation of same instance, instead cloning of existing object and modify as per our need.

Implementation →

* Object which were copying should provide copying feature by implementing cloneable interface.
* we can override clone() method to implement as per our need

```
class Prototype implements Cloneable {
    private List<String> vehicleList;
    public Vehicle() {
        this.vehicleList = new ArrayList<String>();
    }
    public Vehicle(List<String> list) {
        this.vehicleList = list;
    }
    public void insertData() {
        vehicleList.add("Honda Amaze");
        :
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        List<String> temp = new ArrayList<String>();
        for (String s : this.getVehicleList()) {
            temp.add(s);
        }
        return new Vehicle(temp);
    }
}
class Example {
    public static void main() throws CloneNo —— {
        Vehicle a = new Vehicle();
        a.insertData();
        Vehicle b = (Vehicle) a.clone();
```

Builder Design Pattern →

* Used when we have too many arguments to send in Constructor & it's hard to maintain the order.
* When we don't want to send all parameters in Object Initialisation.

Implementation

* We create 'static nested class' containing all arguments of Outer class.
* If classname is 'Vehicle', builder classname : VehicleBuilder
* Builder class → public constructor with only required parameters
* Builder class → optional parameters method with return type Builder Object.
* 'build()' method returns the final object.
* Main class 'vehicle' has private constructor so to create instance via Builder class.
* Main class 'vehicle' has only getters not setters.

```
class vehicle {
    engine, wheel // required
    airbag // optional
    getEngine()
    getWheel()
    getairbag()
    private vehicle(VehicleBuilder builder){
        this.engine = builder.engine ---------- 3.
    public static class VehicleBuilder {
        engine, wheel, airbag ---- // 3 variables
        public VehicleBuilder(String engine, int wheel){
            this.engine = engine ---- 3
        public VehicleBuilder setAirbags(int airbags){
            this.airbags = airbags;
            return this;
    3
```

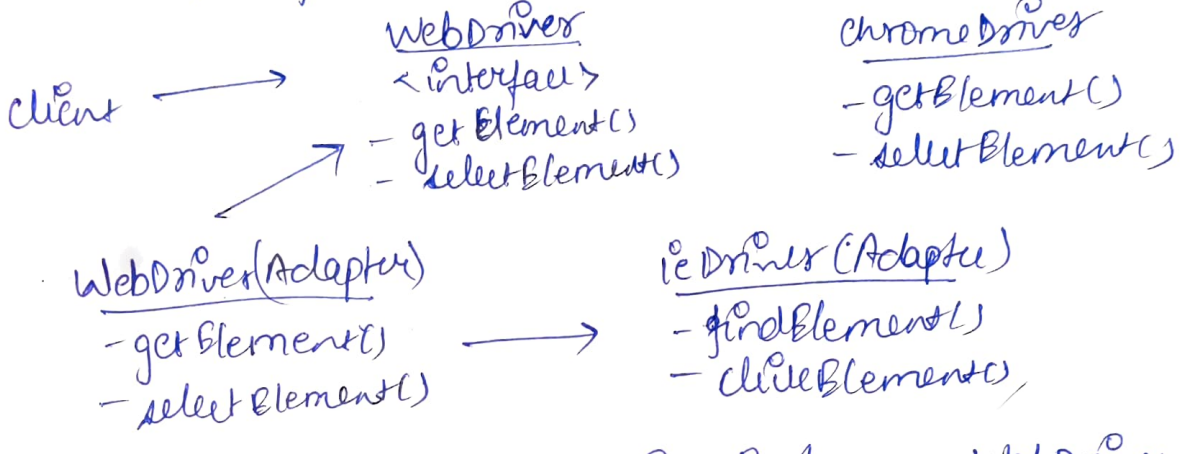
```
        public Vehicle build() {
            return new Vehicle(this);
        }
    }
}

class Example {
    public static void main() {
        Vehicle car = new Vehicle.VehicleBuilder("1500cc", 4).
                      setAirbags(4).build();
        Vehicle bike = new Vehicle.VehicleBuilder("250cc", 2).build()
```

Structural Design Pattern

It simplifies the structure by identifying the relationships. Focus on how classes inherit from each other and how they are composed from ~~of each~~ other. classes.

Adapter Design Pattern → Wrapper

* When objects offering same features, but has different interface
* It allows existing classes to be used with others without modifying source code.
* Converts interface of a class into another interface that a client wants

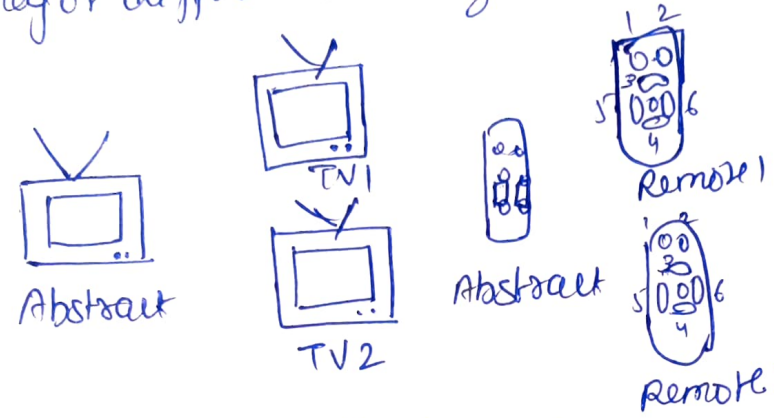



```
interface WebDriver
    getElement();
    selectElement();

ChromeDriver implements WebDriver
    getElement() // sop("chrome");
    selectElement() // sop("chrome");

IEDriver
    findElement() // sop("IEDriver");
    clickElement() // sop("IEDriver");

WebDriverAdapter implements WebDriver
    IEDriver ieDriver;
    getElement()
    {  ieDriver.findElement();
    }
    selectElement()
    {  ieDriver.clickElement();
    }
```

Internally, it is calling IEDriver methods

Bridge Design Pattern →

* Decoupling an abstraction from its implementation so that the two can vary independently
* Progressively adding functionality while separating out major differences using abstract classes.



* 2 layers of abstraction between classes
* 1 class is dependent upon the other

For TV → 5 & 6 will be used to switch channels.
For DVD Player → 5 & 6 is used to skip the video

When to use

* when you want to change both the abstractions (abstract classes) and concrete classes independently.

Entertainment
<<abstract>>
int devicestate, maxsetting;
volumelevel = 0;
abstract fivepress();
abstract sixpress();
.sevenpress(); → volumelevel++
eightpress(); → volumelevel--

TVDevice extends Entertainment
TVDevice (newstate, maxset)
{ devicestate = newstate —
3.
fivepress() { sop("channel Down");
devicestate--;
3
sixpress() { sop("channel Up");
devicestate++;
3

DVDDevice extends Entertainment
DVDDevice (new —, max —)
=
3
fivepress() { sop("skip 10s");
sixpress() { sop("skip chapter");
++;
3

Remote class next page

RemoteButton
<<abstract>>
Pvt. EntertainmentDevice theDevice
RemoteButton(Ent. newDev)
theDevice = newDev.
fire() → theDev.buttonPressed()
set()
abstract buttonPressed();

DVDRemote extends RemoteButton
private play = true;
DVDRemote(Ent. newDev.)
super(newDev)
buttonPressed() { play = !play
sop("Playing" + play);
}

TVRemoteMute extends Remote
TVRemoteMute(Ent. newDe
{ super(newDevice)
}
buttonPressed()
{ sop("Mute")
}
}

TVRemotePause - - -
=
buttonPressed()
{ sop("Pause")
}

Composite Design Pattern →
* It lets client treat individual objects (leaf) and compositions of objects (composite) uniformly.
* four participants: component, Leaf, composite, client
* if object is leaf Node, request is handled directly
if object is composite, it forward to child & perform operations

Implementation
- Component: Account class, which contain common method
- Leaf: DepositeAccount & SavingAccount
- Composite: CompositeAccount
- Client: client class

We'll get balance of all account for a Person.

Account
<<abstract>>
abstract getBalance();

SavingAccount extends Account
- String accountNo;
- float accountBalance;
+ SavingAccount(accNo, acBal)
{ = =
}
+ float getBalance()
{ return accountBalance; }.

DepositeAccount extends Account
- String accountNo;
- float accountBalance;
+ DepositeAccount(acNo, acBal)
// constructor
{ super()
this. — = — ;
}
+ float getBalance()
{ return accountBalance; }.

Client

main()
{
CompositeAcc comp = new Comp—();
comp.addAccount(new Depo-("001", 100));
——— (new Dep—("002", 200);
——— (new Saving-("SA001", 100));
float totalBal = comp.getBalance();
SOP("Balance" + totalBal);
}

CompositeAccount extends Account
- float totalBalance
- List<Account> acclist = new ArrayList<Account>();
+ float getBalance {
totalBalance = 0;
for(Account f : acclist)
{ totalBalance += f.getBalance();
} return totalBalance;
}
+ addAccount(Account acc)
{ acclist.add(acc);
}
+ removeAccount(Account acc)
{ acclist.remove(acc);
}

Decorator Design Pattern → Wrapper

* Used when we want to modify functionality of object at runtime & it should not change individual object functionality.
* More flexible than Inheritance.
i.e. Adding different functionalities in dress.

```
Dress
<<interface>>
+ assemble();

BasicDress implements Dress
{
 + void assemble()
 { sop("Basic Dress features");
}}

Class DressDecorator implements Dress
protected Dress dress;
 + DressDecorator(Dress c)
 { this.dress = c; }
  void assemble()
  { this.dress.assemble(); }
}

CasualDress Impl_ DressDeco-
+ CasualDress(Dress c)
{ super(c); }
void assemble()
{ super.assemble();
  sop("Add casual Dress");
}}

SportyDress implements DressDeco
+ SportyDress(Dress c)
{ super(c); }
+ void assemble()
{ super.assemble();
  sop("Add Sporty Dress");
}}

FancyDress Implem
+ FancyDress(Dress c)
{ super(c); }
+ void assemble
{ super.assemble()
  sop("Add fancy
}}

class clientDesignPattern{
 P S V main()
{ Dress sportyDress = new sportyDress(new BasicDress());
  sportyDress.assemble();
  —
  —
  Dress sportyFancyDress = new SportyFancyDress(new Fancy(new Basic()));
  sportyFancyDress.assemble();

Dress casualFancy = new CasualFancy(new FancyDress(new Basic()));
casualFancy Dress.assemble();
```

Facade Design Pattern

* It hides the complexities of the system and provides an interface to the client which the client can access the system.
* Every Abstract Factory is a type of facade.

Shape
```
<<interface>>
void draw();
```

Rectangle imp. Shape
```
+ void draw()
{ sop("Rectangle"); }
```

Square imp. Shape
```
+void draw()
{ sop("Square"); }
```

Circle imp. Sh
```
+void draw()
{ sop("Circle"
```

ShapeMaker → facade Class
```
- Shape circle, rect, square;
+ ShapeMaker() {
    circle = new Circle();
    rect = new Rectangle();
    square = new Square();
  }
+ void drawCircle()
{ circle.draw(); }
```

```
class facadePatternDemo{
  p s v main()
{
  ShapMaker shapeMaker = new
                ShapeMaker();
  shapMaker.drawRectangle();
  ShapeMaker.drawCircle();
  ShapeMaker.drawSquare();
}
```

Flyweight Design Pattern

* It is primary used to reduce the no. of objects created & to decrease memory footprint & increase performance.
* It tries to reuse already existing similar objects by storing them and creates new object when no matching is found.

Example →

* We will draw 20 circles with 5 colors & different centers
* But only 5 objects are created so this color property is used to check already existing circle objects.

Shape

<<Interface>>

void draw();

ShapeFactory {

Pvt. static final HashMap<String, Shape> map = new HashMap();

+ static Shape getCircle(color) {

Circle circle = (Circle) map.get(color);

if(circle == null) {

circle = new Circle(color);

map.put(color, circle);

Sop("Circle created" + color);

} return circle;

}

}

Circle implements Shape

- String color;
- int x, y, radius;

+ Circle(String color)

{ this.color = color; }

+ setX(); setY(); setRadius();

draw() {

Sop("Circle" + x + y + radius);

FlyweightPatternDemo {

+ static final color[] = {"Red", "Green" --};

P S V main {

for(int i=0; i<20; ++i) {

Circle circle = (Circle) shapeFactory.getCircle(

getRandomColor());

circle.setX(getRandomX());

circle.setY(getRandomY());

circle.setRadius(100);

circle.draw();

}}

+ static String getRandomColor() {

return color[(int)(Math.random() * color.

Length()

+ static int getRandomX() {

return (int)(Math.random() * 100);

+ static int getRandomY() {

return (int)(Math.random() * 100);

Proxy Design Pattern

A class represents functionality of another class.
We create object having original object to interface its functionality to outer world.

Ex →

Image

```
{ <<Interface>>
  void display();
```

ProxyImage implements Image

```
- RealImage real;
- String file;
+ ProxyImage(fileName)
{ this.file = fileName; }
void display()
{ if(real == null)
  { real = new RealImage(file);
    real.display();
  }
}
```

RealImage Implements Image

```
- String fileName;
RealImage(FileName)
{ this.fileName = FileName;
  loadFromDisk(FileName);
}
void display()
{ sop("Displaying" + fileName); }
void loadFromDisk(FileName)
{ sop("Loading" + fileName); }
}
```

ProxyDemo

```
P S V main()
{
Image img = new ProxyImg("abc");
// image loaded from disk
img.display()
// image not loaded this time
img.display();
}
```

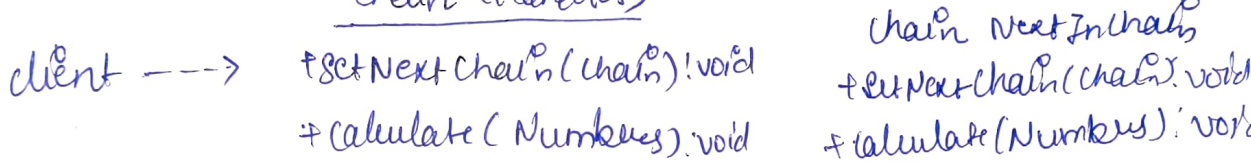
Behavioral Design Pattern

The interaction between the objects should be in such a way that they can easily talk to each other and still be loosely coupled.

Chain of Responsibility

It creates a chain of receiver objects for a request. This pattern sends data to an object and if that object can't use it, it sends it to any number of other objects that may be able to use it.

Ex → Create 4 objects that can either add, sub, multiply or divide send 2 numbers & a command & allow these 4 objects to decide which can handle the requested calculation.



Code →

```
interface Chain {
public void setNext(Chain next);
public void calculate(Numbers nos);
}

class Numbers {
private int num1, num2;
private String calWanted;
Numbers(int n1, int n2, String cal)
{ num1 = n1; ... }
public int getNumber1(){ return num1;}
public int getNumber2(){ return num2;}
public String getCalWanted(){ return cal;}
}
```

```
class Add implements Chain {
private Chain next;
public void setNext(Chain chainnext)
{ this.next = chainnext; }
public void calculate(Numbers request)
{ if(request.getCalWanted()=="add")
{ SOP(req.getNumber1()+req.getNum2()); }
else
{ next.calculate(request); }
```

Same classes for Sub, Multiply and Divide class →

```
Divide class{
P r calculate (Numbers request)
{   if(——)
    {
    }
    else
    { sop("Only works for add, sub, multiply and divide");
    }
}

Class TestChain {

P S V main(){

Chain c1 = new Add();
Chain c2 = new Sub();
=
Chain c4 = new Divide();
c1.setNextChain(c2);
c2.setNextChain(c3);
c3.setNextChain(c4);

Numbers request = new Numbers(4,2,"add");
c1.calculate(request);
}
}
```

Command Design Pattern

A request is wrapped under an object as command and passed to invoker object. Invoker object looks for appropriate object which can handle this command & passes the command to corresponding object which executes the comma

Implementation →

Command → Order (Interface)

Concrete Command → BuyStock & SellStock

Invoker → Broker

Interface Order

```
void execute();
```

Stock class (Properties of stock)

```
String name = "ABC";
int quantity = 10;
void buy(){
  sop("Stock Bought");}
  void sell(){
    sop("stock sold");}
```

BuyStock implements Order

```
Pvt. Stock stock;
P BuyStock(Stock newStock)
{  stock = newStock;
3
  void execute(){
    ~~stock.execute()~~
      stock.buy();
3 3
```

SellStock implements Order

```
Pvt. Stock stock;
P. SellStock(Stock newStock)
{ stock = newStock; }
P void execute(){
  stock.sell(); }
3
```

Broker (Invoker)

```
List<Order> list = new ArrayList<Order>();
P void takeOrder(Order order)
{   list.add(order);
3
P void placeOrder(){
for(Order order : list)
{   order.execute();
  3
  list.clear();
3
3
```

Command Demo

```
P S V main()
{ Stock abc = new Stock();
  BuyStock b = new BuyStock();
  SellStock s = new SellStock();
  Broker br = new Broker();
    br.takeOrder(b);
    br.takeOrder(s);
    br.placeOrder();
3 3
```

Interpreter Pattern →

This pattern provides a way to evaluate language grammar or expression.
Used in SQL Parsing
generally not used due to better solutions.

Iterator Design Pattern:

It is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Ex → Nok - Object class is used to refer any object whose type is unknown

Interface Iterator

```
p boolean hasNext();
p Object next();
```

Interface Container

```
p Iterator getIterator();
```

p. class NameRepo implements Container

```
{ p String names[] = {"Robert", "John", "Lora"};
  p Iterator getIterator()
  { return new NameIterator(); }
  // inner class
  pvt class NameIterator imp. Iterator{
    int index;
    p boolean hasNext();
    { if( index < names.length)
      { return true; }
      else { return false; };
    p Object next(){
      if (this.hasNext()){
        return names[index++];
      }
      return null;
    }}}
```

Iterator Demo

```
{
p s v main(){
NameRepo repo = new NameRepo();
for(Iterator itr = repo.getIterator();
    itr.hasNext();)
{ String name = (String) itr.next();
  sop(name);
}}}
```

Note

java.util. Iterator interface uses Iterator Design Pattern.

Mediator Design Pattern

It is used to reduce communication complexity b/w multiple objects or classes. It provides a mediator class which normally handles all communication b/w different classes.

Ex → <<Mediator Interface>>

Interface chatMediator

```
p void sendMsg(String msg, User user)
  void addUser(User user)
```

<<concrete Mediator>>

chatImp implements chatMediator

```
pvt List<user> users;
chatImp(){ this.users = new ArrayList<>(); }
p v addUser(User user){
 { this.users.add(user); }
p v sendMsg(str msg, User user)
{ for(User u: this.users)
  { if(u != user)
    { u.receive(msg); }
  } } }
class chatClient {
  p s v main()
  { chatMediator mediator = new chatMediator();
    User u1 = new UserImp(mediator, "Abhina");
    User u2 = new UserImp(mediator, "Akshi");
    mediator.addUser(u1);
    ---- (u2);
    u1.send("Hi All");
```

<<User Interface>>

abstract class User

```
prot. chatMediator mediator;
prot. String name;
User(chatMediator med, name)
{ this.— = —
}
p abstract void send(String msg);
p abstract void receive(String msg);
```

<<concrete User>>

UserImp extends User

```
p UserImp(chatMediator med, name)
{ super(med, name); }
p v send(String msg)
{ sop("Sending=" + msg);
  mediator.sendMsg(msg, this);
}
p v receive(String msg);
{ sop("Revived=" + msg);
}
```

Memento Design Pattern → Token

It is used to restore state of an object to previous state

Memento → contains state of an object to be restored.

Originator → creates & stores states in memento Objects

Caretaker → To restore object from Memento.

Memento class

```
pvt. string state;
p  Memento(String state)
{  this.state = state; }
p  String getState()
{  return state; }
}
```

class Originator {

```
pvt. String state;
p v setState(String st)
{ this. ——— }
p v .getState(){ return state; }
p Memento saveStateToMemento(){
 return new Memento(state);
}
p void getStateFromMemento(
                Memento memento)
{ state = memento.getState(); }
}
```

Caretaker class

```
{
pvt. List<Memento> mementoList = new ArrayList<Memento>();
 p v add(Memento state){
    mementoList.add(state);
 }
 p Memento get(int index){
    return mementoList.get(index);
 }
}
```

Memento Demo class

```
{  p s v main(){
   Originator origin = new Originator();
   Caretaker care = new Caretaker();
   origin.setState("State 1");
   care.add(origin.saveStateToMemento());
   origin.setState("State 2");
   origin.getStateFromMemento(care.get(0));
```