CRC – Class Responsibility Collaboration.

Page No.
Date :

*. Types of Relations in Objects:

1. Association:
Has separate / different life cycles. There are more than 2 objects which exists in isolation.

2. Aggregation: Has same Ownership relation.
– different life cycles of objects.
– loose binding. (Parent – Child).

3. Composition: Tight binding between the Owner and Association.

***. SOLID PRINCIPLE:
Solid are five basic principles which help to create good quality software.

S – Single Responsibility Principle:
Which says that the class should have only one responsibility. If a class has more than one responsibility, then break the parent class into the modules.

O – Open / Closed Principle:
It states that classes should be open for extension but closed for

modification.

L – Liskov substitution Principle:

If we create a derived class for a base class, it should properly reflect the base class and extend it without replacing the functionality of old classes.

I – Interface Segregation Principle: –

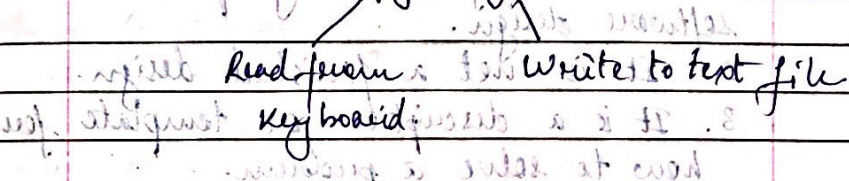
In an interface, the methods which are declared will be compulsorily implemented by the classes implementing it, but there will be times when some classes needs only some of the methods from the interface. So interface segregation is important.

D – Dependency Inversion Principle:

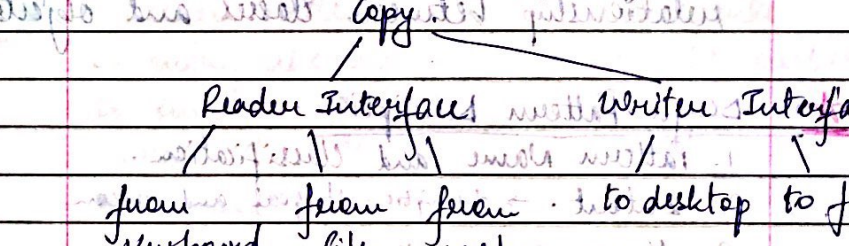
Here, the problem is the functionality should be dependent on the abstraction and not vice versa, because the flexibility of the software will go down and we wont be able to add some other functionality to it.

Base class reference can point to any of the derived class objects.

Ex: (i) Without DIP:



(ii) With DIP:



In (i), if you want to read from file the whole abstract will change, i.e., abstract is getting dependant on the detail

In (ii), Reader is a bread interface – indipend – ant of details.

If you want to add details, Ex: reading from a file (a detail) is dependant on the abstract – Reader.

P.T.O.

Design Pattern < Class Level / Object Level

Page No.
Date :

* DESIGN PATTERNS:

1. It is a general reusable solution to a commonly occurring problem in software design.
2. It is not a finished design.
3. It is a description or template for how to solve a problem.
4. Object Oriented Design Patterns show relationship between classes and objects.

*** Design Pattern Description

1. Pattern Name and Classification.
2. Intent. → description of goal and reason.
3. Known as → Other names.
4. Motivation (Forces) → A scenario consisting of problem.
5. Applicability.
6. Structure.
7. Partition.
8. Collaboration → classes and objects interaction
9. Consequences.
10. Implementation
11. Sample code.
12. Known Use.
13. Related Patterns → Other patterns which can be used with in combination.

It is possible that a system can be designed using different design patterns.

Page No.
Date :

** Design Pattern Space : (Diagram).

* Classification of Design Patterns:

1. Creational DP.
2. Structural DP.
3. Concurrency Behavioural DP.
4. Concurrency Design Pattern.

** Design Patterns V/s. Frameworks.

- ↳ more abstract
- ↳ smaller architectural elements.
- ↳ cannot be directly implemented into a coding language.

- ↳ can be embedded in code.

** CREATIONAL DESIGN PATTERN:

- ↳ All about class instantiation.
- ↳ 2 Categories:
  - Class Creational Pattern – effective inheritance
  - Object Creational Pattern – effective delegation
- ↳ Focuses on abstraction rather than structure.

** STRUCTURAL DESIGN PATTERN:

- ↳ All about Class and Object composition.
- ↳ Structural class-creation patterns use inheritance to compose interfaces.
- ↳ Structural object patterns define ways to

Factory method is a method in an abstract class, whose child classes will have to override those functions.

compose objects to obtain new functionality.

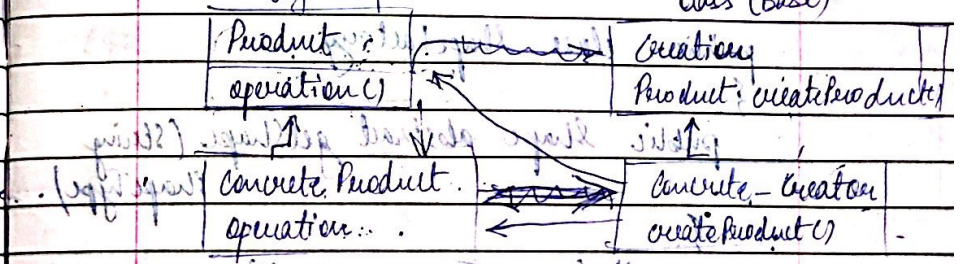
** BEHAVIOURAL DESIGN PATTERN:
↳ All about object classes, object communication
↳ Most specifically concerned with communication between objects

** Concurrency DP:
↳ About multi-threaded paradigm.

1. Creational Patterns:
↳ Abstracts instantiation process.
↳ Make system independant of how its objects are
→ created
→ composed
→ represented

■ Factory Method: (Virtual constructor):
• Defer object instantiation to subclasses.
• Eliminates binding of application specific subclasses
• Connects parallel class hierarchies
• A related pattern is AbstractFactory.
• Factory method Example (Diagram):

Define an interface for creating an object, but let the subclasses decide which class to instantiate. Lets object creation defer to subclasses.

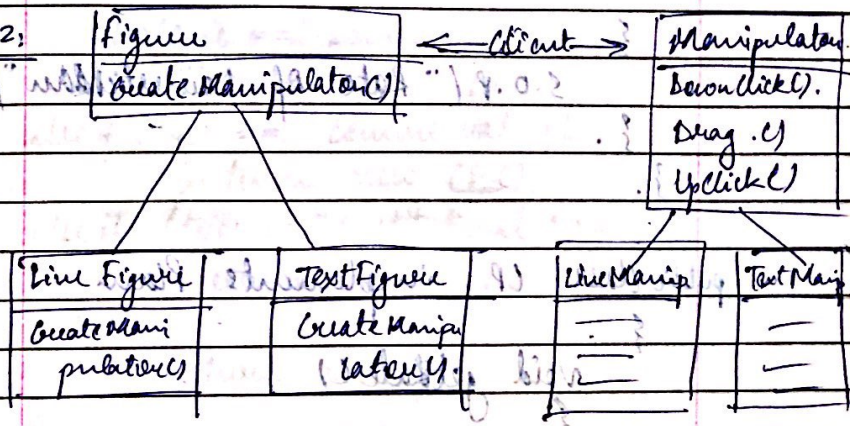


Ex: Shape Example:

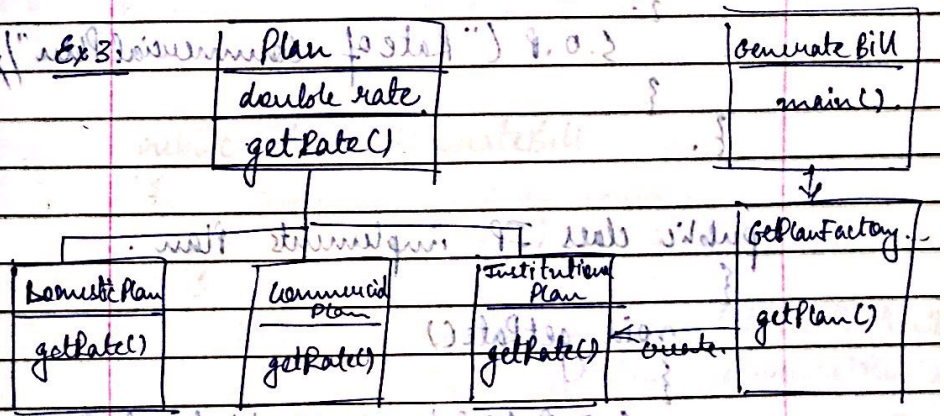
```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    public void draw() {
        System.out.println("Circle Drawn");
    }
}

public class Square implements Shape {
    public void draw() {
        System.out.println("Square Drawn");
    }
}
```

```
public class ShapeFactory
{
  public Shape abstract getShape (String
                                  ShapeType).
  {
    if (shapeType == null)
        return null;
    if (shapeType.equalsIgnoreCase("circle"))
    {
      return new circle();
    }
    else if (ShapeType.equalsIgnoreCase("square"))
    {
      return new square();
    }
    return null;
  }

public class FactoryMain
{
  psvm (String[] a)
  {
    ShapeFactory sf = new ShapeFactory();
    Shape s = sf.getShape("circle");
    s.draw();
  }
}
```

Ex 2:



Ex 3:



Soln:

```
public interface Plan
{
  double rate;
  void getRate();
}

public class DP implements Plan
{
  void getRate()
```

```
{
    S.O.P("Rate of DomesticPlan");
}
public class CP implements Plan
{
    void getRate()
    {
        S.O.P("Rate of CommercialPlan");
    }
}
public class IP implements Plan.
{
    void getRate()
    {
        S.O.P("Rate of Institutional Plan");
    }
}
public class GetPlanFactory.
{
    public Plan abstract getPlan(String type)
    if (type == null)
        return null;
    if (type == "DomesticPlan")
        return new DP();
    else if (type == "CommercialPlan")
        return new CP();
    else if (type == "InstitutionalPlan")
        return new IP();
    else
        return null;
}
public class GenerateBill
{
    p.s.v.m (S a[])
    {
        GetPlanFactory gpf = new GetPlanFactory();
        Plan p = gpf.getPlan("CommercialPlan");
        p.getRate();
    }
}
```

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

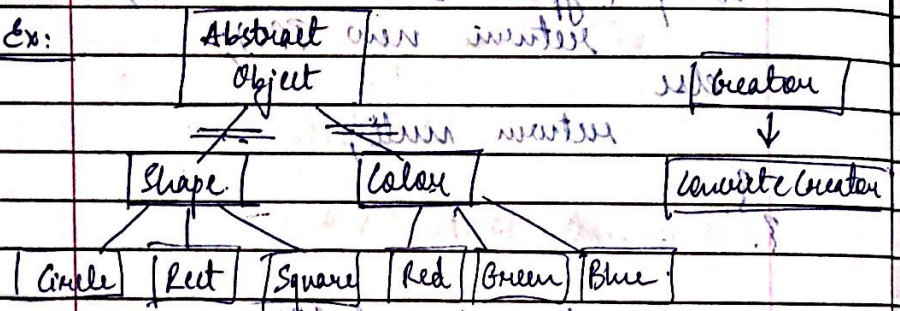
★ Abstract Factory Pattern: (Kit).

↳ Define an interface or abstract class

↳ Here when you want to have more than one type of hierarchies, then you may have to create more than ~~factory methods~~ parent Generic Class

↳ Define an interface or abstract class for creating families of related or dependant objects but without specifying their concrete sub-classes.

Ex:



Soln: abstract class AbstractObject

```
{
<T> getObjectType (String s)
{
if (s.equalsIgnoreCase("shape")
{
return new Shape();
}
else
{
return new Color();
}
}
```

```
abstract class Shape
{
abstract void draw();
Shape getShape (String s)
{
if (s == "null")
return null;
else if (s == "square")
return square();
else if (s == "rectangle")
return rectangle();
else
return circle();
}
}
class Circle extends Shape.
{
void draw()
```

Ex: Computer Hierarchy Example:

```
public class Computer.
{
public abstract String getRAM();
public abstract String getHDD();
public abstract String getCPU();
}
```

```
public class PC extends Computer
{
}
public class Server extends Computer{;
{
}
public interface ComputerAbstractFactory
{
    public Computer createComputer();
}
public class PCFactory implements CAF
{
    public Computer createComputer()
    {
        return new PC();
    }
}
public class ServerFactory implements CAF
{
    public Computer createComputer()
    {
        return new Server();
    }
}
public class ComputerFactory
{
    public static Computer getComputer(CAFactory
    }
    return factory.createComputer();
}
public class TestDesignPatterns
{
    public(s[3])
    {
        testAbstractFactory();
    }
    private static void testAbstractFactory()
    {
        Computer pc = ComputerFactory.getComputer();
        Computer server = ComputerFactory.getComputer
        ();
    }
}
```

* Applicability: Abstract Factory:

1. A system should be independent of how its products are created, composed, and represented.
2. When there several kinds of product objects, and they must be designed to network together.

* Difference between Abstract FP and Builder.

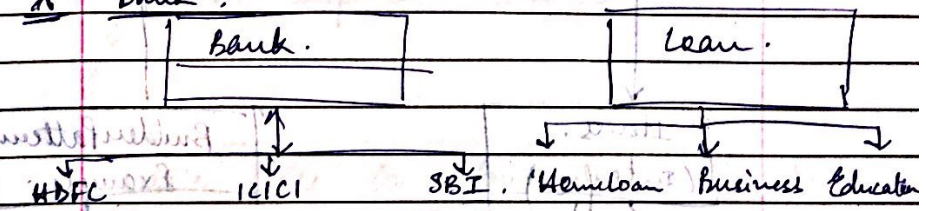
Main for Shape and Color:

```
AbstractObject a = FactoryGenerator.getFactory("SHAPE");
Shape s = a.getShape("CIRCLE");
s.draw();
a = FactoryGenerator.getFactory("COLOR");
Color sl = a.getColour("RED");
sl.fill();
```

```
public abstract class AbstractFactory
{
    abstract Colour getColour(String colour);
    abstract Shape getShape(String shape);
}

class FactoryGenerator
{
    public static AbstractFactory getFactory
        (String fT)
    {
        if (fT == null)
            return null;
        else if (fT.equalsIgnoreCase("SHAPE"))
            return new ShapeFactory();
        else if (fT.equals("COLOUR"))
            return new ColourFactory();
        else return null;
    }
}
```

FactoryGenerator — returns AbstractFactory.
decides ShapeFactory or ColourFactory.

* Bank:



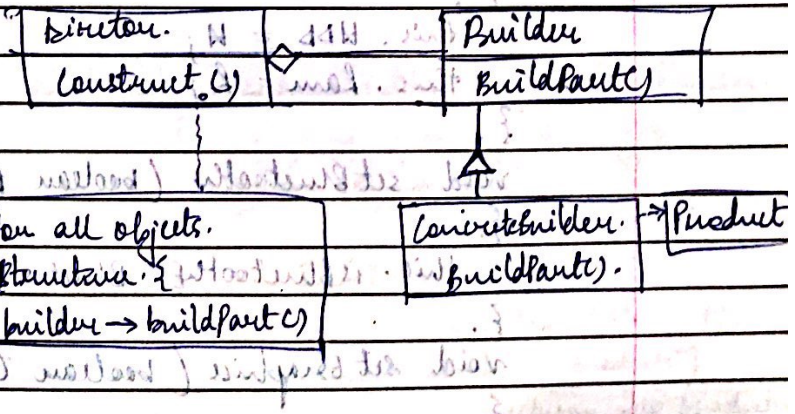
** BUILDER PATTERN:

— Construct a complex object from simple objects using step-by-step approach.

Intent:

Separate the construction of a complex object from its representation so that the same construction pattern can create different representations

Structure:



Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Ex: Meal (A menu consist of Items) — MealBuilder

Meal → Items (Interface) → BigMac, IceCream, Tea, Cocacola

MealBuilder → BuilderPattern Example

↳ Meal is the product class

Ex: Computer Hierarchy: (Available on Net — Tousmal Dey)

```
public class Computer
{
    private String HDD, Ram, isBluetooth, isGraphicsE;

    private Computer (String H, String R)
    {
        this.HDD = H;
        this.Ram = R;
    }
    void setBluetooth (boolean Bluetooth)
    {
        this.isBluetoothE = Bluetooth;
    }
    void setGraphics (boolean Graphics)
    {
        this.isGraphicsE = Graphics;
    }
}

public static class ComputerBuilder. // Nested class
{
```

Ex: Person

Person → Student, Employee

Student → Graduate, Postgraduate

Employee → Temporary, Permanent

1. Factory Pattern:

```
abstract class Person.
{
    abstract void details();
}

abstract class student extends Person.
{   String name, id, cgpa;
    void details()
    {
        S.o.p("");
    }
    void type (String t).
    {
        if(t=="Graduate")
            return new Graduate();
        else return new Postgraduate();
    }
}
```

```
class Employee extends Person.
{
    String name, id, salary;
    void details()
    {
        S.O.P.(" ");
    }
}

class Graduate extends Student.
{
    Graduate(String n, String i, String c)
    {
        name = n;
        id = i;
        cgpa = c;
    }
    void details()
    {
        S.O.P.(name + id + cgpa);
    }

class PostGraduate extends Student
{
    PostGraduate(String n, String i, String c)
    {
        name = n;
        id = i;
        salary = c;
    }
    void details()
    {
        S.O.P.(name + id + cgpa);
    }

abstract class PersonFactory (continued)
    public Person getPerson(String type)
    {
        if (type == "Student")
            return new Student();
        if (type == "Employee")
            return new Employee();
    }

class CreatePerson
{
    public static void main(String[] a)
    {
        PersonFactory pf = new PersonFactory();
        Person p = pf.getPerson("Student");
```

+ public
− private
protected
} class diagram specifiers.

Person p1 = p.getType("Graduate");
p1.details();
}
}

4. Singleton Design Pattern:

1. Define a class that has only one instance and provides a global point of access to it.
2. A class must ensure that only single instance should be created and single object can be used by all other classes.
3. Two Forms:

Static 1. Early Instantiation:
Creation of instance at load time.

Dynamic 2. Lazy Instantiation:
Creation of instance when required.

4. It is used where only a single instance of a class is required to control the execution throughout the program execution.
5. For a single object creation, use private constructor.

* Structure:

Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData

→ return unique instance

Ex: public class Singleton1
{
private int id;
private String name, college;
private static Singleton1 single = new Singleton1();
private Singleton1() { }
public static Singleton1 getInstance()
{
return single;
}
public void setInstance(int i, String n, String c)
{
this.id = i;
this.name = n;
this.college = c;
}

Scanned with CamScanner

```
public void display()
{
    System.out.println("Id is " + id + "Name is " + name + "College is " + college);
}
}.

public class SingletonDP.
{
    public static void main(String[] args)
    {
        S.O.P("Hello Singleton Pattern");
        Singleton object = Singleton.getInstance();
        Object.setInstance(12, "Abc", "RCOEM");
        object.display();
    }.
}.
```

5. ~~Prototype Design Pattern~~

Singleton Design Pattern Benefits:

1. Control access to sole instance.
   The Singleton class encapsulates its sole instance, can have strict control over how and when client access it.

2. It reduces namespace:
3. Permits refinement of operations and representations.

5. Prototype Design Pattern:

↳ Required when object creation is time consuming and costly operation.
↳ We create object with existing object itself.
↳ allows us to hide the complexity of making new instances from the client.
↳ The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations.
↳ The existing object acts as a prototype and contains the state of the object.
↳ The newly copied object may change some properties only if required.
↳ This approach saves costly resources and time, especially when the object creation is a heavy process.

Example:

```
class1 obj1 = new class1();
class1 obj2 = new class1();
```

obj1.stCode = "old value";
obj2 = obj1;
obj2.stCode = "New value";
messagebox.show("value of obj1 = "+obj1.stCode
+"value of obj2 ="+obj2.stCode);

Output:

obj1	New value
obj2	New value

→ When we change property of another referencing object, the original object's property is changed. To avoid this we use clone() method in java.

Example using clone:

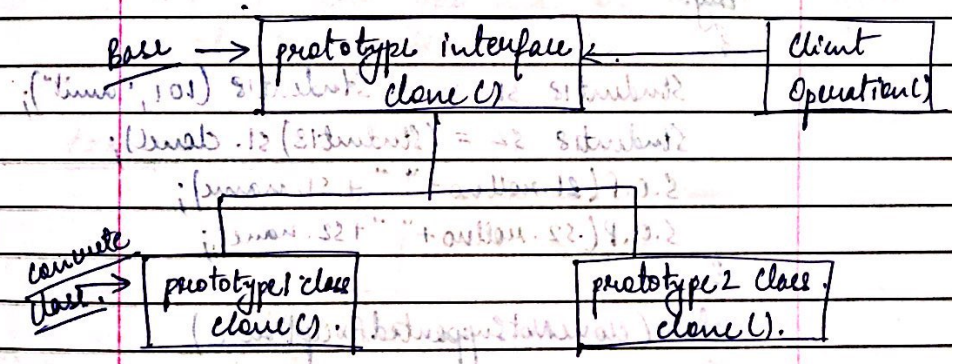
```
public class Customer
{
    public String customerCode = "";
    public String customerName = "";
    public Customer getClone()  //used in
    {
        return (Customer) this.memberwise
                                clone();
    }
}
```

↑ client create the object of customer class.

* Cloning is always by value not by reference.

```
Customer obj1 = new Customer();
obj1.customerCode = "002";
obj1.customerName = "old customer";
Customer obj2;
obj2 = obj1.getClone();
```

* Structure:



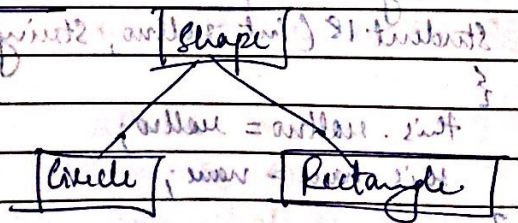
Example:

```
class Student18 implements Cloneable.
{
    int rollno;
    String name;
    Student18(int rollno, String name)
    {
        this.rollno = rollno;
        this.name = name;
    }
}
```

P.T.O.

```
public Object clone() throws CloneNotSupportedException
{
    return super.clone();
}

public static void main(String args[]){
try
{
    Student18 s1 = new Student18(101, "amit");
    Student18 s2 = (Student18) s1.clone();
    S.O.P(s1.rollno + " " + s1.name);
    S.O.P(s2.rollno + " " + s2.name);
}
catch(CloneNotSupportedException c)
{
    c.printStackTrace();
}
}
```

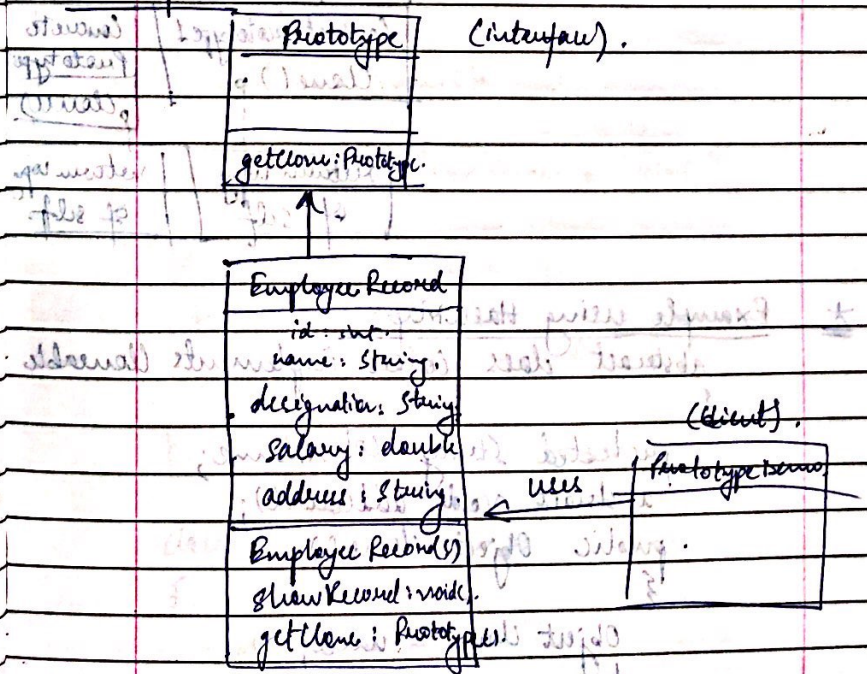
H.W:



in Laptop

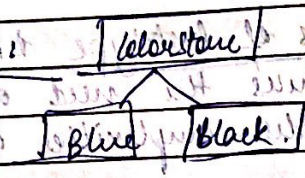
- Advantages of Prototype DP:
  1. It reduces the need of sub-classing.
  2. It hides complexities of creating objects.
  3. The clients can get new objects without knowing which type of object it will be.
  4. It lets you add or remove objects at runtime.

Example:

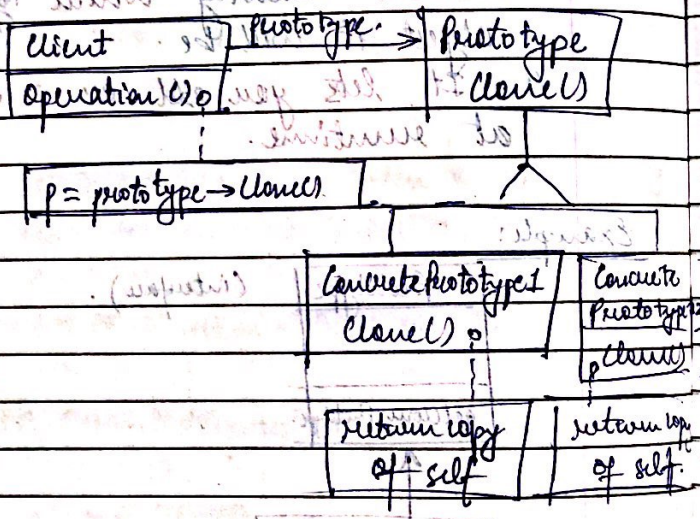


in Laptop.

• ColourStore:



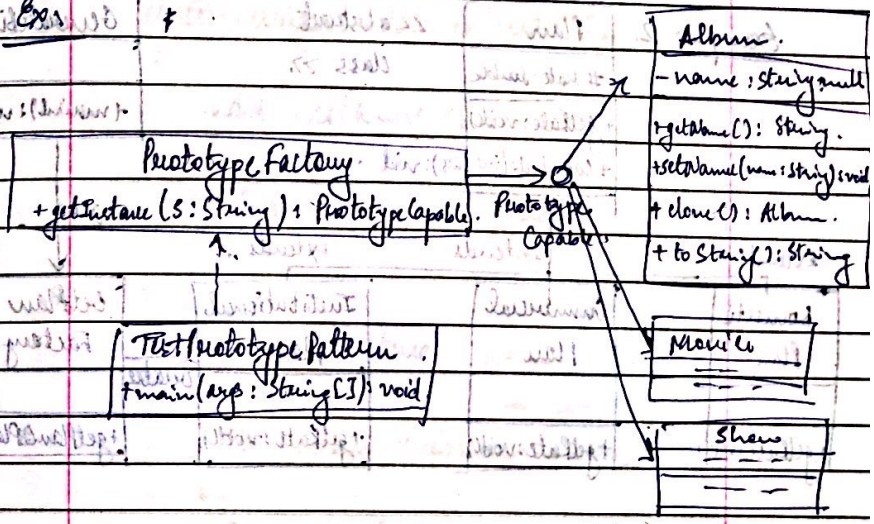
★ Structure:



★ Example using HashMap:

```
abstract class Colour implements Cloneable
{
protected String colourName;
abstract void addColour();
public Object clone()
{
Object clone = null;
try
{
clone = super.clone();
}
catch (CloneNotSupportedException c)
{
c.printStackTrace();
}
return clone;
}
}
class BlueColour extends Colour
{
public BlueColour()
{
this.colourName = "Blue";
}
@Override
public void addColour()
{
sout("Blue Colour Added");
}
}
class BlackColour extends Colour
{
public BlackColour()
{
this.colourName = "Black";
```

```
@Override
public void addColour()
{
    Sout("Black Colour Added");
}
}

// Prototype
class ColourStore
{
    private static Map<String, Colour> colourMap
    = new HashMap<String, Colour>();
    static {
        colourMap.put("blue", new BlueColour());
        colourMap.put("black", new BlackColour());
    }
    public static Colour getColour(String colourName)
    {
        return (Colour) colourMap.get(colourName).clone();
    }
}
class DriverMain // Client.
{
    psvm()
    {
        ColourStore.getColour("blue").addColour();
        ColourStore.getColour("black").addColour();
        ColourStore.getColour("black").addColour();
        ColourStore.getColour("blue").addColour();
    }
}
```

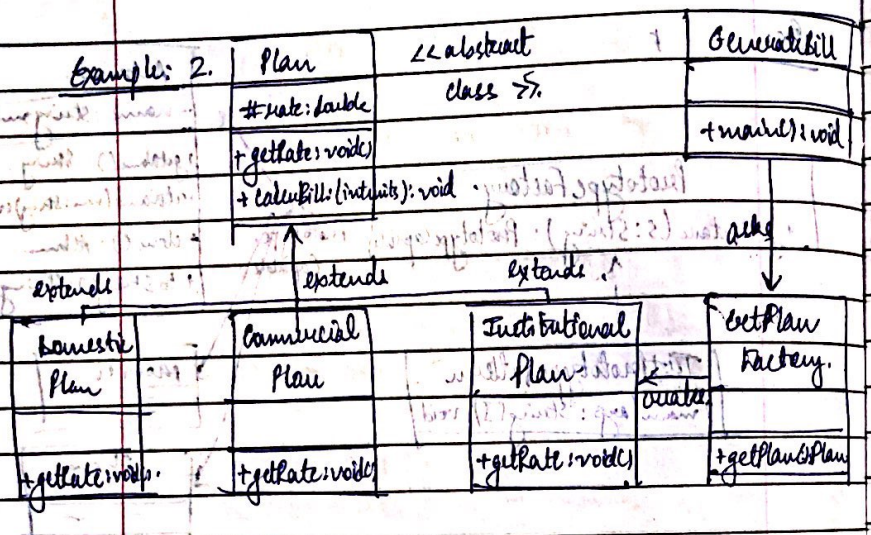


★ REVISION:

1. FACTORY PATTERN:
   - ↳ Related to object creation.
   - ↳ Multiple implementation of same interface.
   - ↳ Hide logic of object creation from user/client.

Eg 1: Factory → create → OS (interface / abstract class)
Client → ask → Factory
OS: Android, Windows, IOS.

Example: 2.



Ex 1: Code:

```
abstract class OS.
{.
   String osname;
   abstract void setName();
   void display()
   {.
      sout(osname);
   }.
};
class AndroidOS extends OS
{
   void setName()
   {
      osname = "Android OS";
   }.
}.
```

```
class WindowsOS extends OS.
{.
   void setName()
   {.
      Osname = "Windows OS";
   }.
};
class IOS extends OS
{
   void setName()
   {.
      Osname = "IOS";
   }.
}.
class Factory.
{.
   OS.
   static void getOSType(String type).
   {.
      if (type == null)
         return null;
      else if (type.equalsIgnoreCase("Android"))
         return new AndroidOS();
      else if (type.equalsIgnoreCase("Windows"))
         return new WindowsOS();
      else if (type.equalsIgnoreCase("IOS")
         return new IOS();
      else
         return null;
   }.
}.
```

```
public class OSmain
{
public static void main(String args[])
{
  Scanner sc = new Scanner(System.in);
  System.out.println("Enter OS type");
  String type = sc.next();
  OS os = Factory.getOSType(type);
  os.display.setName();
  os.display();
}
}
```

UNIT — 3.

STRUCTURAL DESIGN PATTERNS

↳ These are concerned with how classes and objects can be composed, to form larger structures.

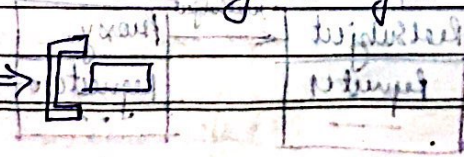
↳ These simplifies the structure by identifying the relationships.

↳ These patterns focus on, how the classes inherit from each other and how they are composed from other structures.

- Types:
  1. Adapter
  2. Bridge.
  3. Composite
  4. Decorator.
  5. Façade.
  6. Flyweight.
  7. Proxy.

Proxy Design Pattern:

Intent: Lets you provide a substitute or placeholder for another object. A proxy controls the access to the original object, allowing you to perform something either before or after the request gets through to the original object.



* To identify Proxy DP in a code given, the object will be initialized to null and checks `if(object == null)`.

↳ Proxies are also called surrogates, handles and wrappers.

↳ Literal meaning is — 'in place of', 'representing', 'on behalf of', etc.

* ↳ Controls and manages access to the object they are protecting.

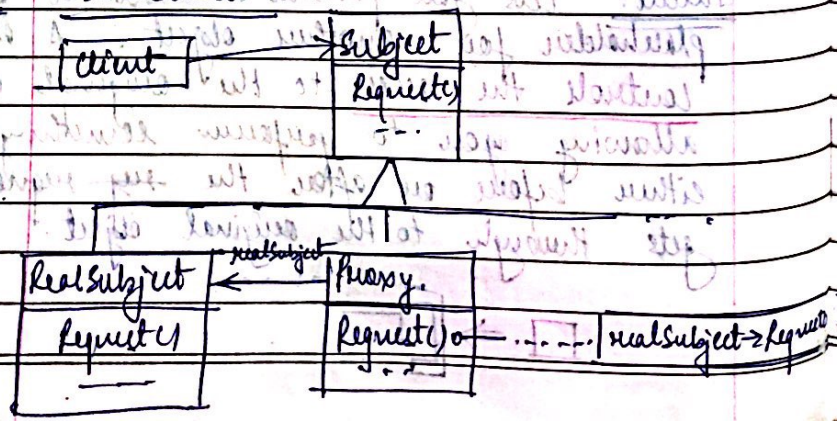
* **Advantages:**

1. Security.
2. Avoids duplication of objects. This in turn increases the performance of the application.
3.

* **Applicability:**

1. Remote Proxy.
2. Virtual Proxy.
3. Protection Proxy.
4. Smart Reference / Smart pointers.

* **Structure:**



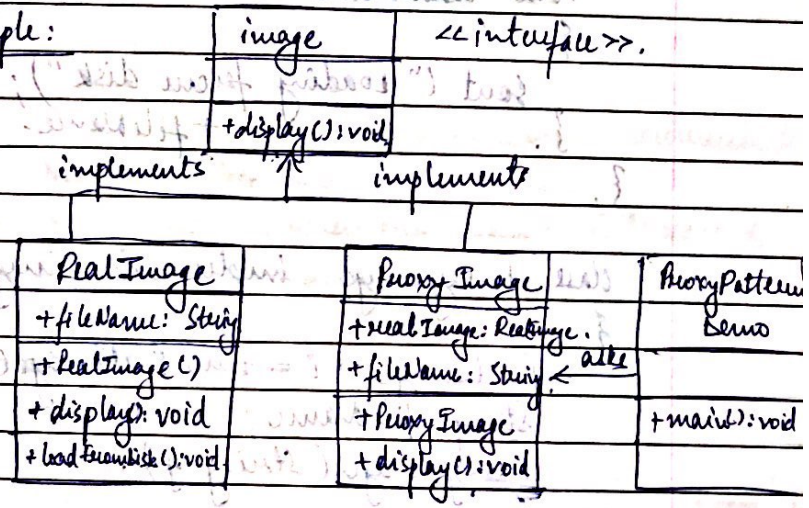
* **Participants:**

↳ Proxy.

↳ Subject — common interface.

↳ Real Subject. — original Object.

Example:



Solution:

```
public interface Image
{
public void display();
}

public class RealImage implements Image.
{
    public String fileName;
    RealImage(String f)
    {
        fileName = f;
        loadFromDisk();
    }
```

```
void display()
{
    Sout("display the image");
}
void loadFromDisk()
{
    Sout("loading from disk" + fileName);
}

class ProxyImage implements Image
{
    RealImage ri = new RealImage();
    String fileName;
    ProxyImage(String f)
    {
        fileName = f;
        ri = new RealImage(fileName);
    }
    void display()
    {
        ri.loadFromDisk();
        ri.display();
    }
}
if(ri==null)
ri=new RealImage(fileName)

class ProxyPatternDemo
{
    psvm(s a[])
    {
        ProxyImage pi = new ProxyImage("c:\...");
        pi.display();
    }
}
```

- 1. Proxy improve the structure communicate with interface
- 2. Proxy class implements same interface to control access / access control.

```
main from Main's:
class ProxyPattern
{
    psvm()
    {
        Image image = ProxyImage("test.jpg");
// Image loaded from disk
        image.display();
        Sout("");
// Image not loaded from disk
        image.display();
    }
}
```

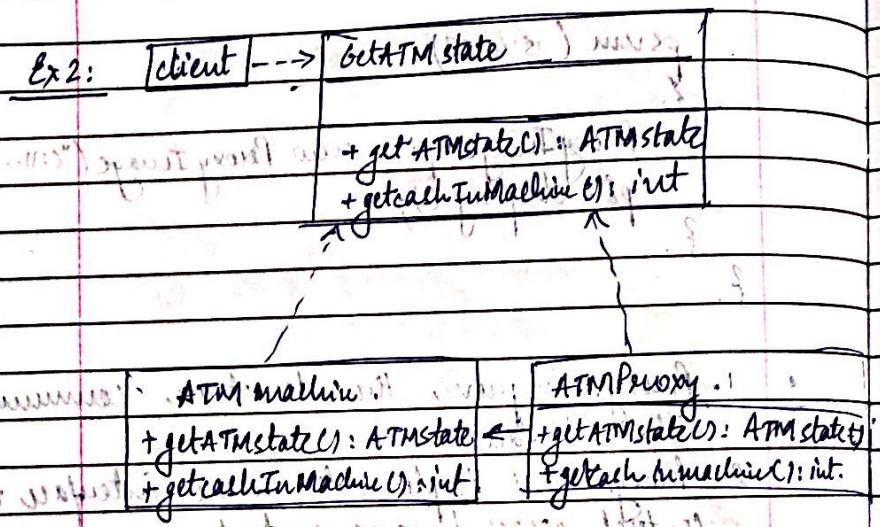
O/P:

loading test.jpg

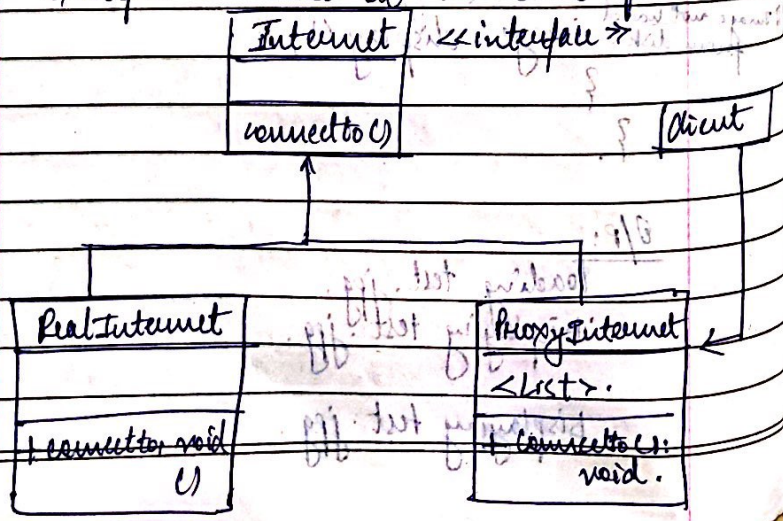
Displaying test.jpg

Displaying test.jpg

Ex 2:



Ex 3: A very simple real life scenarios is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.




```
public interface Internet
{
    void connectto(String domain);
}

class RealInternet implements Internet
{
    void connectto(String domain)
    {
        Sout("connecting to "+domain);
    }
}

class ProxyInternet implements Internet
{
    ArrayList<String> banned = new ArrayList();
    RealInternet ri;
    void connectto(String domain)
    {
        if (banned.contains(domain))
        {
            Sout("Access denied");
        }
        else
        {
            if (ri == null)
                ri = new RealInternet();
```

Scanned with CamScanner

```
static
{
banned.add("abc.com");
banned.add("efg.com");
}.
ri.connectto(domain);
}.
}.
}.
class client
{
psvm(S args[])
{
ProxyInternet pi = new ProxyInternet();
i.connectto("www.google.com");
}
}.
```

*** Decorator Design Pattern:

Also known as Wrapper

- Intent:

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing to extend flexibility.

↳ The structure of the original object is not altered, just new responsibilities are added.

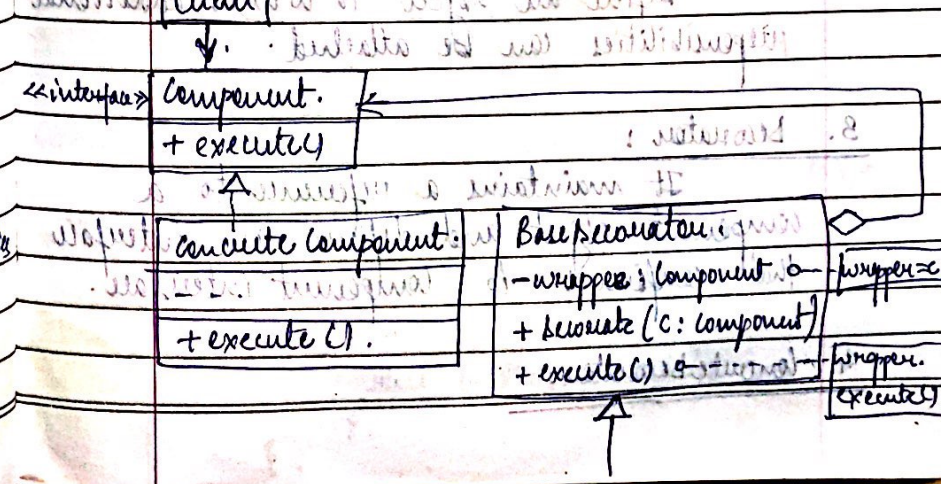
↳ Decorator is a conceptual pattern that allows adding new behaviours to objects dynamically by placing them into special wrappers.

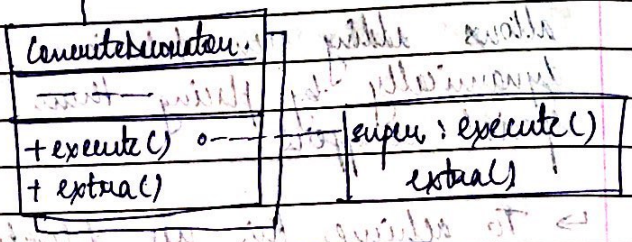
↳ To achieve this use delegation.

Inheritance: Inheritance in Java programming is the process by which one class takes the property of another class.

Delegation: It can be viewed as a relationship between objects where one object forwards certain methods calls to another object, called its delegate.

* Structure:






```
a = new ConcreteComponent()
b = new ConcreteDecorator1(a)
c = new ConcreteDecorator2(b)
c.execute()
```

- Participants:

4 Participants :-

1). Component (Visual Component):

Define the interface for objects that can have responsibilities added to them dynamically.

2. Concrete Component:

Define an object to which additional responsibilities can be attached.

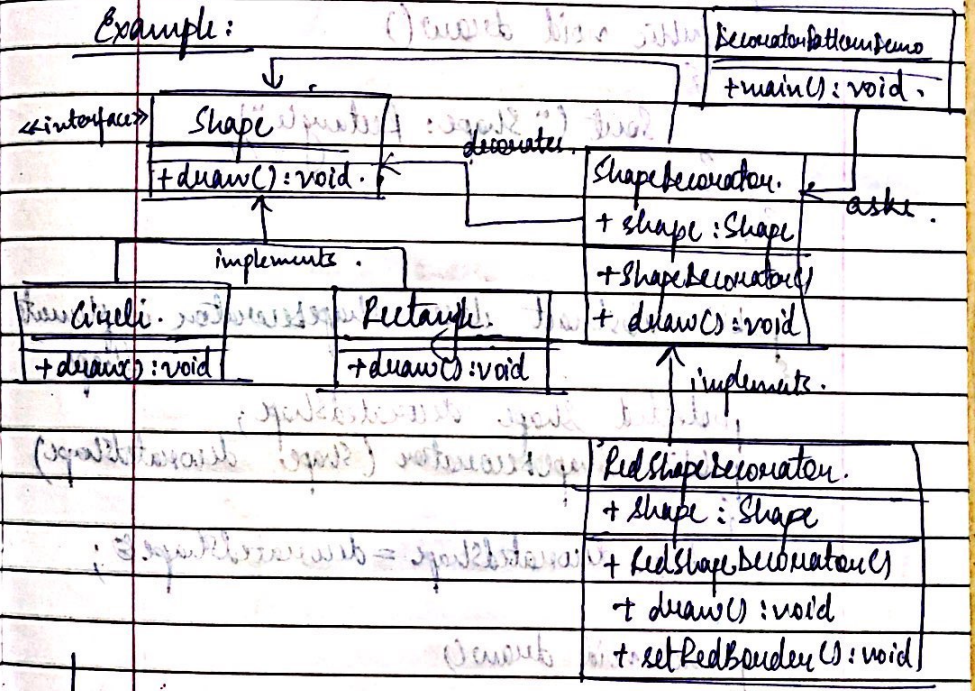
3. Decorator:

It maintains a reference to a component object and defines an interface that conforms to Component interface.

4. Concrete Decorator:

It adds responsibilities to the component.

Example:

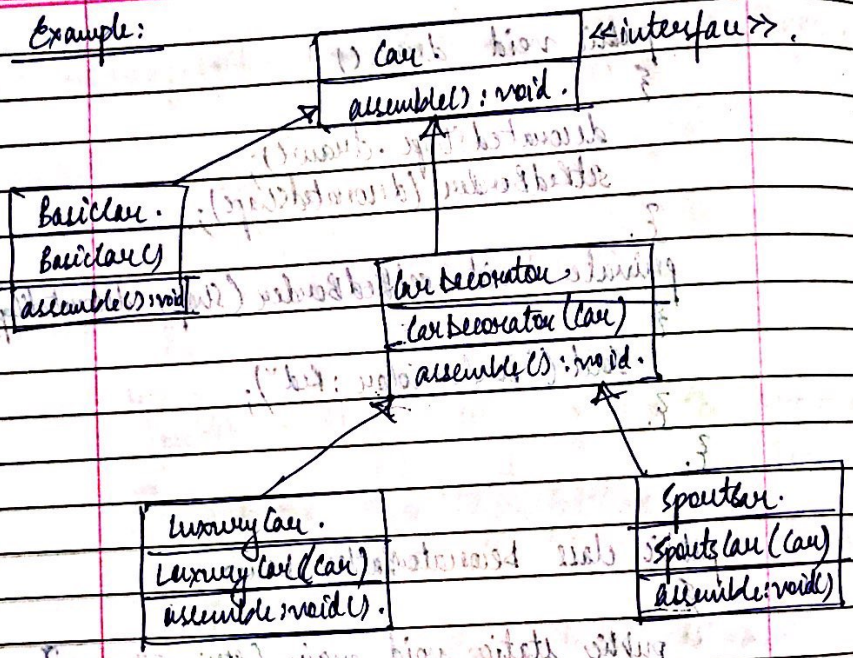


Example:

```
public interface Shape
{
    void draw();
}
public class Circle implements Shape.
{
    public void draw()
    {
        Sout("Shape: Circle");
    }
}
```

```
public class Rectangle implements Shape.
{
    public void draw()
    {
        Sout ("Shape: Rectangle");
    }.
}.

public abstract class ShapeDecorator implements
                                        Shape.
{
    protected Shape decoratedShape;
    public ShapeDecorator (Shape decoratedShape)
    {
        this. decoratedShape = decoratedShape();
    }.
    public void draw()
    {.
        decoratedShape. draw();
    }.
}.

public class RedShapeDecorator extends ShapeDecor.
{
    public RedShapeDecorator (Shape decoratedShape
    {
        super (decoratedShape);
    }.
    public void draw()
    {
        decoratedShape.draw();
        setRedBorder (decoratedShape);
    }.
    private void setRedBorder (Shape decoratedShape
    {
        Sout ("Border Colour: Red");
    }.
}.

public class DecoratorPatternDemo
{.
    public static void main (String[] args)
    {
        Shape circle = new Circle();
        Shape redCircle = new RedShapeDecorator
                                (new Circle());
        Shape redRectangle = new RedShapeDecorator
                                (new Rectangle());
        Sout ("Circle with normal Border");
        circle. draw();
        Sout ("Circle of Red Border");
        redCircle. draw();
        Sout ("Rectangle of Red Border");
        redRectangle. draw();
    }.
}.
```

Example:



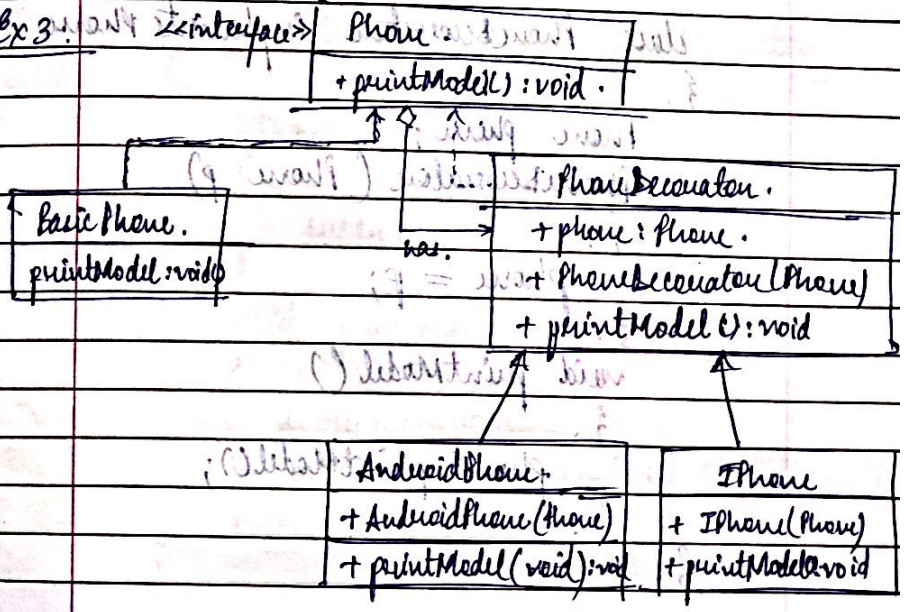
Code:

```
public interface Car.
{
    void assemble();
}
public class BasicCar implements Car.
{
    BasicCar()
    {
    }
    void assemble
    {
        Sout("Assembling Basic Car");
    }
}
```

```
public class CarDecoration implements Car.
{
    protected Car car;
    public CarDecoration(Car c)
    {
        this.car = c;
    }
    public void assemble()
    {
        this.car.assemble();
    }
}

public class SportsCar extends CarDecoration.
{
    public SportsCar(Car c)
    {
        super(c);
    }
    public void assemble()
    {
        super.assemble();
        Sout("Adding features of Sports Car");
    }
}

public class LuxuryCar extends CarDecoration.
```

```
public LuxuryCar(Car c)
{
    super(c);
}
public void assemble()
{
    super.assemble();
    sout("Adding features of Luxury Car");
}
}

class demo
{
    psvm(S a[])
    {
        Car sportscar = new SportsCar(new BasicCar());
        sportscar.assemble();
        Car sportsluxury = new SportsCar
        (new LuxuryCar(new BasicCar()));
        sportsluxuryCar.assemble();
    }
}
```

Ex 3:



Solution:

```
public interface Phone
{
    public void printModel();
}

class BasicPhone implements Phone
{
    public void printModel()
    {
        sout("Basic Phone is created");
    }
}
```

```
class PhoneDecorator implements Phone
{
    Phone phone;
    PhoneDecorator(Phone p)
    {
        phone = p;
    }
    void printModel()
    {
        phone.printModel();
    }
}

class AndroidPhone extends PhoneDecorator
{
    AndroidPhone(Phone p)
    {
        super(p);
    }
    void printModel()
    {
        super.printModel();
        Sout("Adding features of Android Phone");
    }
}
```

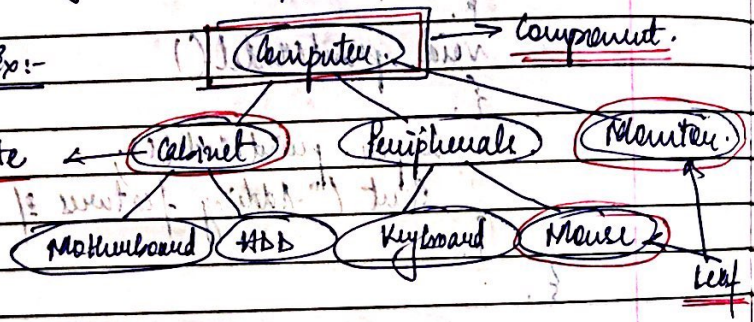
```
class IPhone extends PhoneDecorator
{
    IPhone(Phone p)
    {
        super(p);
    }
    void printModel()
    {
        super.printModel();
        Sout("Adding features of IPhone");
    }
}

class demo
{
    public static void main(String args[])
    {
        Phone iphone = new IPhone(new BasicPhone());
        iphone.printModel();
        Phone mix = new AndroidPhone(new IPhone(new BasicPhone()));
        mix.printModel();
    }
}
```

C = component.
L = Leaf.

Page No.
Date :

** Composite Design Pattern :
↳ A structural pattern which modifies the structure of an object.
↳ Suitable in case where scenario is objects which form a tree like hierarchy.

Eg :-



* Component is the common interface
↳ This pattern helps treat components elements as individual objects and compositions uniformly.

- Participants :
  1. Client
  2. Component. — Root Node.
  3. Composite
  4. Leaf.

- Intent : 1. Compose objects into tree structures to represent a part-whole hierarchies.

Common operation of leaf and composite.

* Group of objects need to be represented as a single object — Use composite design pattern.

Page No.
Date :

2. Composite lets clients treat individual objects and compositions of objects uniformly.

*Q) Demonstrate the participants of composite design pattern.

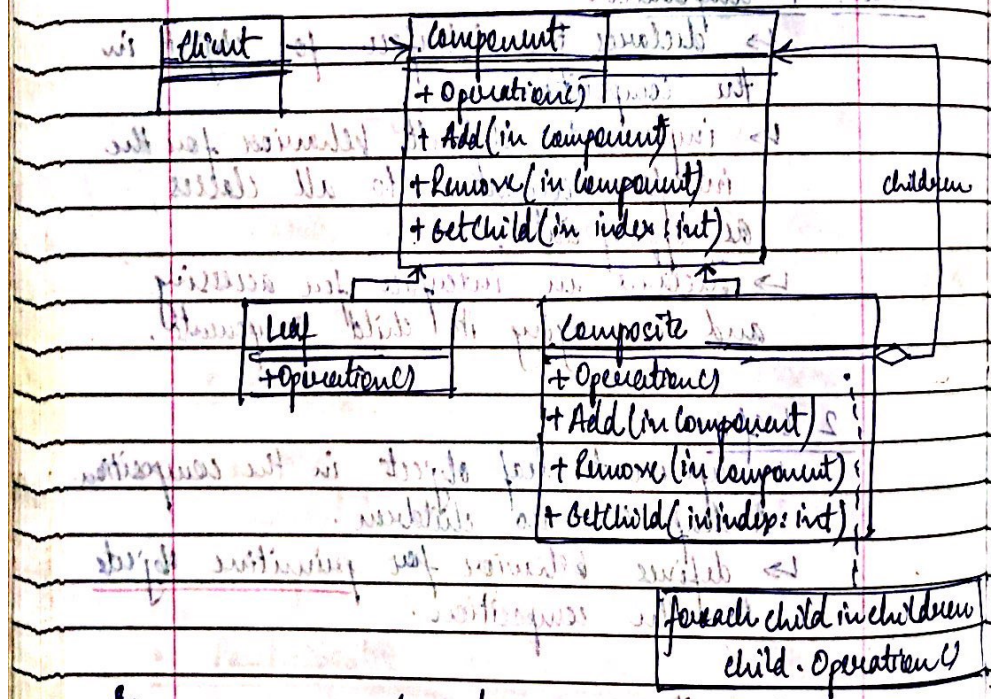
Soln : 1. Component :
↳ declares the interface for objects in the composition.
↳ implements default behavior for the interface common to all classes, as appropriate.
↳ declares an interface for accessing and managing its child components.

2. Leaf :
↳ represents leaf objects in the composition. A leaf has no children.
↳ defines behavior for primitive objects in the composition.

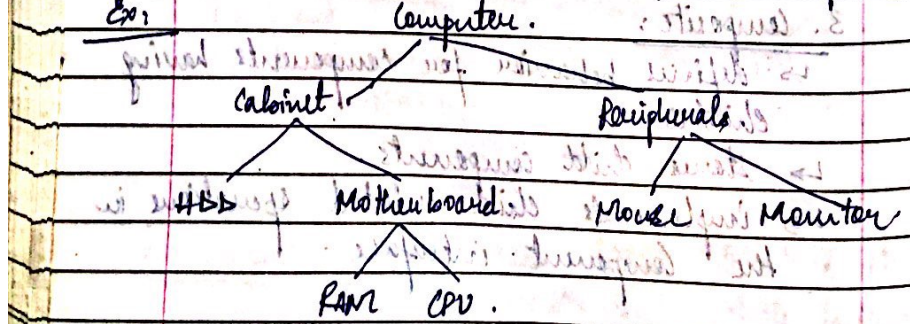
3. Composite :
↳ defines behavior for components having children.
↳ stores child components.
↳ implements child-related operations in the Component interface.

4. Client:
↳ manipulates objects in the composition through the component interface.

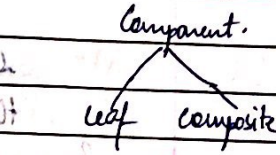
* Structure:



Eg:



Solution:




```
interface Component
{
    void showprice();
}

class Leaf implements Component
{
    int price;
    String name;
    public Leaf(int price, String name)
    {
        super();
        this.price = price;
        this.name = name;
    }
    public void showprice()
    {
        System.out.println(name + ":" + price);
    }
}

class Composite implements Component
{
    List<Component> components = new ArrayList<>();
    String name;
    Composite(String name)
```

Null = Proxy. DP
List = Composite DP.

```
super();
this.name = name;
}
public void addComponent(Component com)
{
components.add(com);
}
public void showprice()
{
System.out.println(name);
for(Component c : components)
{
c.showprice();
}
}
}
public class client
{
psvm(s a[])
{
Component hdd = new Leaf(4000, "HDD");
Component mouse = new Leaf(400, "Mouse");
Component monitor = new Leaf(8000, "Monitor");
Component ram = new Leaf(3000, "RAM");
Component cpu = new Leaf(1000, "CPU");
```

* Composite maintains an array list which holds its children's objects.

```
Composite ph = new Composite("Peripheral");
Composite cb = new Composite("Cabinet");
Composite mb = new Composite("Motherboard");
Composite comp = new Composite("Computer");

ph.addComponent(mouse);
ph.addComponent(monitor);
mb.addComponent(cpu);
mb.addComponent(ram);
cb.addComponent(hd);
cb.addComponent(mb);
comp.addComponent(cb);
comp.addComponent(ph);

System.out.println("Price of all component");
comp.showprice(); sout("Price of all peripherals");
ph.showprice();
}
}
```

O/P:

Price of all component.
Computer:
Cabinet.
HDD : 4000
Motherboard
CPU : 1000
RAM : 3000.

Peripheral

Mouse : 400

Monitor : 8000

Price of all Peripherals

Peripheral

Mouse : 400

Monitor : 8000

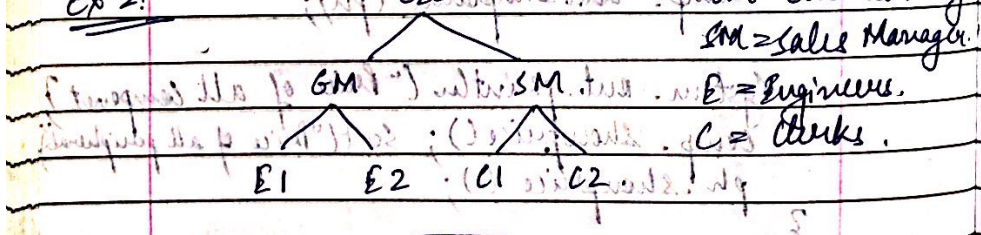
Price of all Motherboard

Motherboard

CPU : 4000

RAM : 8000

Ex 2:



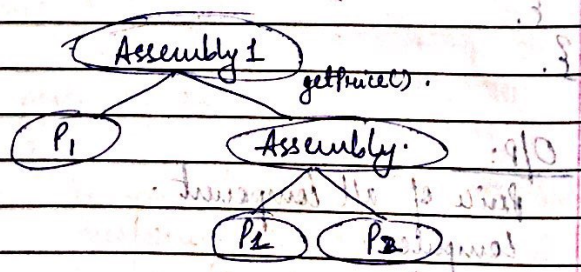
GM = General Manager

SM = Sales Manager

E = Engineers.

C = Clerks.

Ex 3:



Solution:

```
interface Component
{
    void getPrice();
}.
```

```
class Leaf implements Component.
{.
    String name; int price;
    Leaf (String name, int price)
    {
        this.name = name;
        this.price = price;
    }.
    public void getPrice()
    {.
        Sout("name+" :"+price);
    }.
}.
```

```
class Composite implements Component.
{.
    String name;
    List<Component> elements = new
                        ArrayList<>();
    Composite (String name)
    {.
        super();
        this.name = name;
    }.
    public void addComponent (Component com)
    {
        elements.add(com);
    }
```

```
public void showPrice()
{
    Sout("name");
    for (Component c : elements)
    {
        c.showPrice();
    }
}

public class Client
{
    psvm(S a[])
    {
        Component p1 = new Leaf("P1", 1000);
        Component p2 = new Leaf("P2", 2000);
        Component p3 = new Leaf("P3", 1500);

        Component assembly = new Composite("Assembly");
        assembly.addComponent(p2);
        assembly.addComponent(p3);

        Component assembly1 = new Composite("Assembly1");
        assembly1.addComponent(p1);
        assembly1.addComponent(assembly);
        Sout("Total Price of Assembly1 is" +
            assembly1.showPrice());
    }
}
```

- Method 2 using Abstract class:

```
//Assembly
public abstract class Component
{
    float price;
    protected List<Component> list = new
                                ArrayList<>();
    public abstract float getPrice();
    public void add(Component c)
    {
        list.add(c);
    }
    public void remove(Component c)
    {
        list.remove(c);
    }
    public Component getChild(int i)
    {
        return (Component) list.get(i);
    }
}

//Assembly
public class CompositePart extends Component
{
    float total;
    public float getPrice()
    {
        total = 0;
```

```
for (Component f : list)
{
    total = total + f.getPrice();
}
return total;
}
}

public class Part1 extends Component
{
    Part1 (float p)
    {
        price = p;
    }
    public float getPrice()
    {
        return price;
    }
    public void setPrice (float p)
    {
        price = p;
    }
}

public class Part2 extends Component
{
    // same as above class.
}
```

```
public class Composite1
{
    psvm (String a[])
    {
        Component assembly = new CompositePart();
        Component p1 = new Part1 (300);
        Component assembly1 = new CompositePart();

        assembly.add (new Part1 (100.0f));
        assembly.add (new Part2 (500.0f));

        assembly1.add (assembly);
        assembly1.add (p1);
        Sout ("Total price of unit part : "+p1.getPrice());
        Sout ("Total price of assembled part :
                    + assembly1.getPrice());
    }
}
```

O/P:

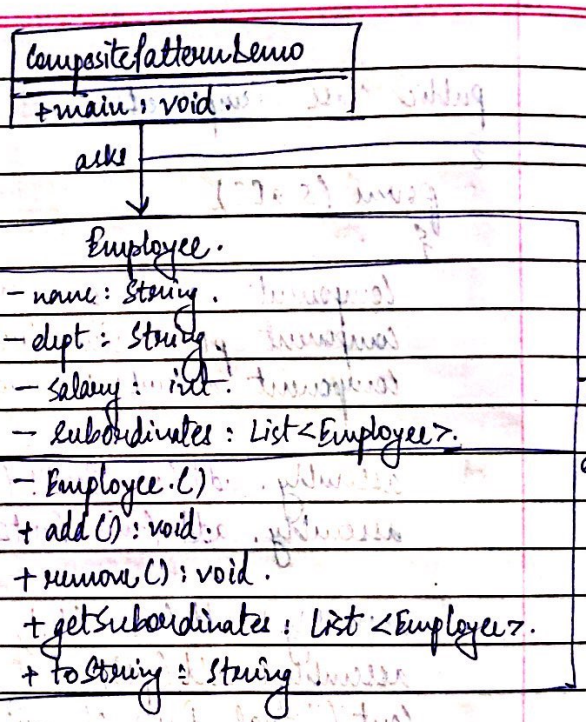
Total price of unit part : 300.0.
Total price of Assembled part : 900.0.

P.T.O.

Composite or Adapter: for 5 marks.

100%.

Ex 3:



Solution: // Only root exists.

```
class Employee
{.
    private String name, dept;
    -n- int salary;
    -n- List<Employee> subordinates
            = new ArrayList<>();
    private Employee (String n, String d, int s)
    {
        name = n;
        dept = d;
        salary = s;
    }.
```

Adapter = Bridge = Mediator = Wrapper.

* Target Interface.

```
public void add (String n, String d, int s)
{.
    tis.subordinates.add(new Employee(n,d,s));
}.
}.
```

*** Adapter Design Pattern: — also called WRAPPER.

Intent:

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

↳ Bridge between two incompatible interface/ classes (also known as "Mediator").

• Adapter:

1. Class — Multiple Inheritance.
2. Object —

↳ Adapter pattern works as a bridge between two incompatible interfaces.

↳ This pattern combines the capability of two independant interfaces.

↳ It involves a single class which is responsible to join functionalities of independant or incompatible interfaces.

↳ Ex: Card read acts as an adapter between

= Adapter Design Pattern:

memory card and laptop.

Intent: An Adapter Pattern says that just "converts the interface of a class into another interface that a client wants".

↳ Adapter can not only convert data into various formats but can also help objects with different interfaces collaborate.

* 1. In design, adapters are used when we have a class (Client) expecting some type of object and
2. We have an object (Adaptee) offering the same features but exposing a different interface.

* Participants in Adapter ~~deg~~ design pattern:

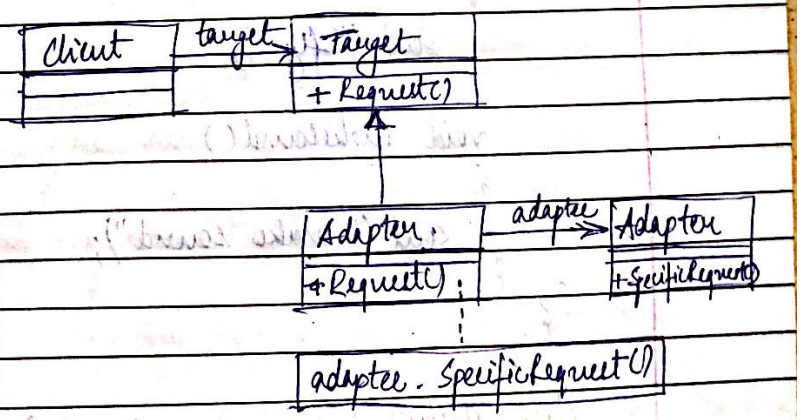
1. Target Interface:
This is the desired interface class which will be used by the clients.

2. Adapter Class:
This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

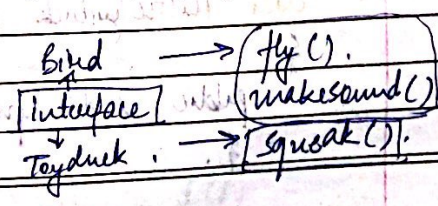
3. Adaptee Class:
This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.

4. Client:
This class will interact with the Adapter class.

* Object Adapter:
* ↳ Adapter holds the instance of Adaptee.
↳ Can be implemented by any programming language.



• Example:
Client → Toyduck.
Adaptee → Bird.



Solution:

```
interface Bird {
    void fly();
    void makeSound();
}

interface ToyDuck {
    void squeak();
}

class Sparrow implements Bird {
    void fly()
    {
        sout("fly");
    }
    void makeSound()
    {
        sout("make sound");
    }
}

class PlasticToyDuck implements ToyDuck {
    public void squeak()
    {
        sout("squeak");
    }
}
```

```
class BirdAdapter implements ToyDuck {
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        this.bird = bird;
    }
    public void squeak()
    {
        bird.makeSound();
    }
}

class AdapterObject
{
    public static void main(String a[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck td = new PlasticToyDuck();
        // Wrap a bird in a BirdAdapter so it will believe like a toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);
        sout("Sparrow..");
        sparrow.fly();
        sparrow.makeSound();
    }
}
```

```
Sout("Toyduck--");
td.squeak();
Sout("BirdAdapter ...");
birdAdapter.squeak();
}
}.
```

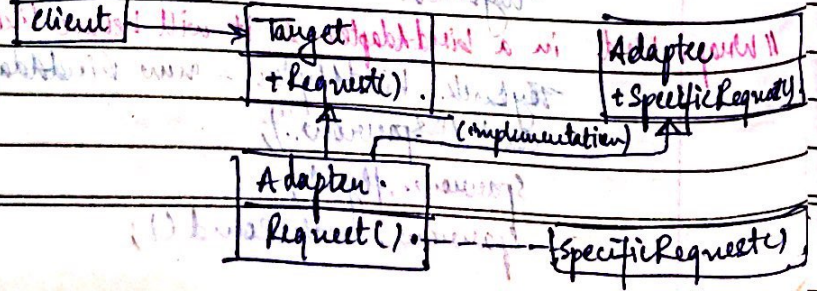
O/P: Sparrow:
Fly.
makeSound.
Toyduck --
Squeak.
BirdAdapter ---
makeSound.

★ Class Adapter:

★ ↳ This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.

↳ Class Adapter pattern which use inheritance instead of composition but you require multiple inheritance to implement it.

• Structure:



Real Life Example:

Ex: Mobile Charger acts as an adapter. Mobile battery needs 3 volts to charge but the normal socket produces either 120v (US) or 240v (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120v).

Ex:

Client
main()

Volt Example:

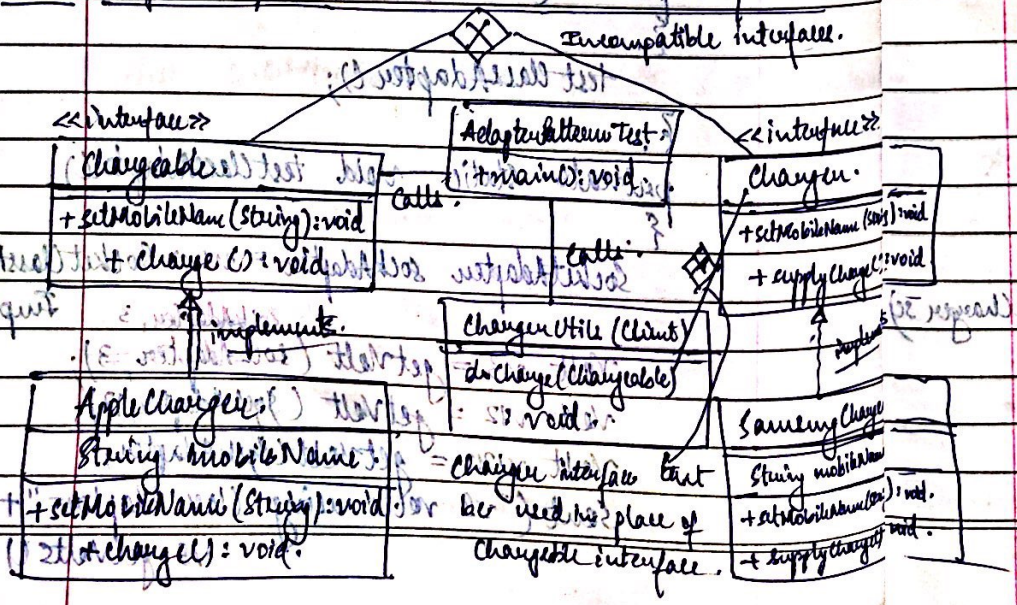
```
public interface SocketAdapter
{
    public Volt get120-Volt();
    public Volt get12-Volt();
    public Volt get3-Volt();
}.

public class Socket
{.
    public Volt getVolt()
    {.
        return new Volt(120);
    }.
}.
```

```
class Volt
{
    private int volts;
    public Volt(int v)
    {
        this.volts = v;
    }
    public int getVolts()
    {
        return volts;
    }
    public void setVolts(int v)
    {
        volts = v;
    }
}

class SocketClassAdapterImpl extends Socket implements SocketAdapter
{
    public Volt get120_Volt()
    {
        return getVolt();
    }
    public Volt get12_Volt()
    {
        Volt v = getVolt();
        return convertVolt(v, 10);
    }
    public Volt get3_Volt()
    {
        Volt v = getVolt();
        return convertVolt(v, 40);
    }
    private Volt convertVolt(Volt v, int i)
    {
        return new Volt(v.getVolts() / i);
    }
}

class AdapterPatternTest
{
    public static void main(String[] args)
    {
        testClassAdapter();
    }
    private static void testClassAdapter()
    {
        SocketAdapter sockAdapter = new SocketClassAdapterImpl();
        Volt v3 = getVolt(sockAdapter, 3);
        Volt v12 = getVolt(sockAdapter, 12);
        Volt v120 = getVolt(sockAdapter, 120);
        System.out.println("v3 volts using ClassAdapter=" +
            v3.getVolts());
```

Scanned with CamScanner

```
sout("V12 volts using Class Adapter = " + v12.getVolts());
sout("V120 volts using Class Adapter = " + v120.getVolts());
}
private static Volt getVolt(SocketAdapter sA, int i)
{
  switch(i)
  {
    case 3: return sA.get3_volt();
    case 12: return sA.get12_volt();
    case 120: return sA.get120_volt();
    default: return sA.get120_volt();
  }
}
}
```

Ex: Object Adapter as it contains 2 interfaces.




```
Solution: interface ChargeableInterface
{
  public void setMobileName(String mN);
  void charge();
}
class AppleCharger implements ChargeableInterface
{
  String mobileName;
  public void setMobileName(String mN)
  {
    this.mobileName = mN;
  }
  public void charge()
  {
    sout(mobileName + " is charging with Apple Charger");
  }
}

interface ChargerInterface
{
  void setMobileName(String mName);
  void supplyCharge();
}
class SamsungChargerImpl implements ChargerInterface
{
  String mobileName;
```

```
public void setMobileName(String mobileName)
{
    this.mobileName = mobileName;
}
public void supplyCharge()
{
    System.out.println("Supplying charge to " + mobileName
        + " through Samsung Charger");
}
class ChargerUtil
{
    void doCharge(Chargeable c)
    {
        c.charge();
    }
}

class AppleAdapter
{
    SamsungCharger sc;
    void getSamsungCharger(SamsungCharger sc)
    {
        this.sc = sc;
    }
    void chargeSamsung()
    {
        sc.supplyCharge();
        System.out.println("Charging through Apple Adapter");
    }
}

class TestDesignPattern
{
    psvm(S.aCI)
    {
        AppleCharger ac = new AppleCharger();
        ac.setMobileName("IPhone");
        ac.charge();
        SamsungCharger sc = new SamsungCharger();
        AppleAdapter aa = new AppleAdapter();
        aa.getSamsungCharger(sc);
        aa.chargeSamsung();
    }
}
```

★ Pros and Cons:

1. Single Responsibility Principle.
2. Open/Closed Principle.
3. Overall Complexity.

- Advantages:

1. Help achieve reusability and flexibility.

Decorator vs. Adapter.

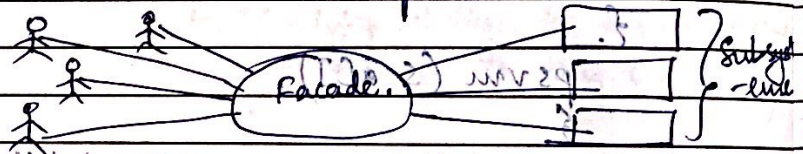
Wrapper
1. Add functionality
2. Single interface

Wrapper.
2. Not adding functionality.
2. Multiple interfaces.

- Disadvantage:
1. All requests are forwarded, so there is a slight increase in the overhead.

** Façade Design Pattern:

↳ Hides all the complexities.



↳ Intent: Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

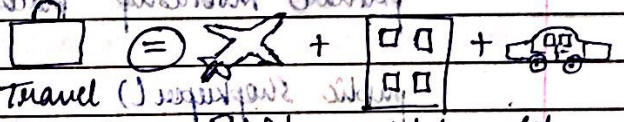
↳ Practically every AbstractFactory is a type of Façade.

- Participants for Façade Design Pattern:

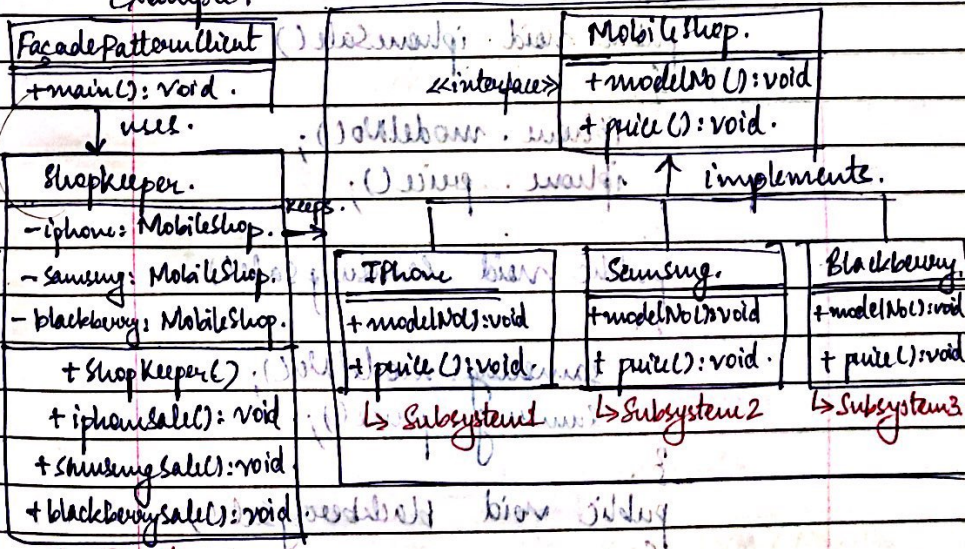
1. Façade:
↳ It knows which subsystem classes are responsible for a request.
↳ Delegates client requests to appropriate subsystem objects.

2. Subsystem classes:
↳ It implements subsystem functionality.
↳ Handle work assigned by the Façade object.
↳ It has no knowledge of the Façade.

Ex:



Example:



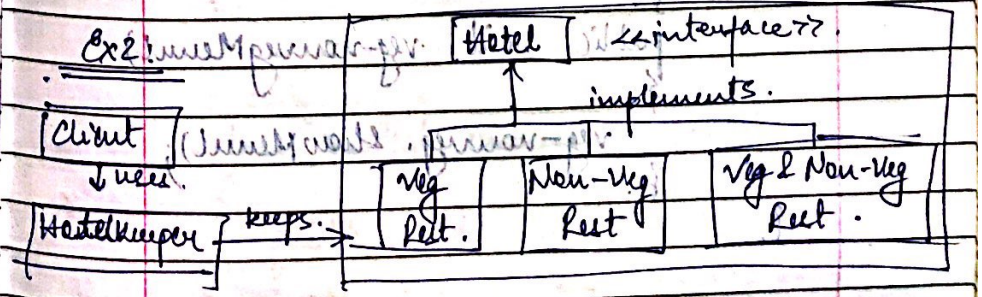
P.T.O

Code:

```
class Shopkeeper
{
    private Mobileshop iphone;
    private Mobileshop samsung;
    private Mobileshop blackberry;

    public Shopkeeper()
    {
        iphone = new Iphone();
        samsung = new Samsung();
        blackberry = new Blackberry();
    }

    public void iphonesale()
    {
        iphone.modelNo();
        iphone.price();
    }

    public void samsungsale()
    {
        samsung.modelNo();
        samsung.price();
    }

    public void blackberrysale()
    {
        blackberry.modelNo();
        blackberry.price();
    }
}
```

```
public class Client
{
    psvm(S.a[])
    {
        int choice = new Scanner.nextInt();
        Shopkeeper sk = new Shopkeeper();
        switch(choice)
        {
            case 1: sk.iphonesale();
            case 2: sk.samsungsale();
            case 3: sk.blackberry-sale();
            default: Sout("invalid choice");
        }
    }
    // define/implement all the subsystems.
```

Ex2:



```
class HotelKeeper
{
    private Hotel veg;
    private Hotel nonveg;
    private Hotel veg-nonveg;
    public HotelKeeper()
    {
        veg = new VegRest();
        nonveg = new NonVegRest();
        veg-nonveg = new VegNonVegRest();
    }
    public void vegMenu()
    {
        veg.showMenu();
    }
    public void nonvegMenu()
    {
        nonveg.showMenu();
    }
    public void veg-nonvegMenu()
    {
        veg-nonveg.showMenu();
    }
}

class Client.
```

```
{
    psvm (String a[]).
    {
        int ch = new Scanner().nextInt();
        HotelKeeper hk = new HotelKeeper();
        switch (ch):
        {
            case 1:
                hk.vegMenu();
            case 2:
                hk.nonvegMenu();
            case 3:
                hk.veg-nonvegMenu();
            default: sout("Invalid Choice");
        }
    }
}

interface Hotel
{
    public void showMenu();
}
class VegRest implements Hotel
{
    public void showMenu()
    {
        sout("Veg Menu items");
    }
}
```

```
class NonVegRest implements Hotel
{
    public void showMenu()
    {
        System.out.println("NonVeg Menu Items");
    }
}
class VegNonVegRest implements Hotel.
{
    public void showMenu()
    {
        System.out.println("Veg & NonVeg Menu Items");
    }
}
```

★ BRIDGE DESIGN PATTERN:

↳ Bridge between two set of hierarchies.

↳ Implements through:

1. Abstraction (Abstract) — class.
2. Implementation (Interface) — interface

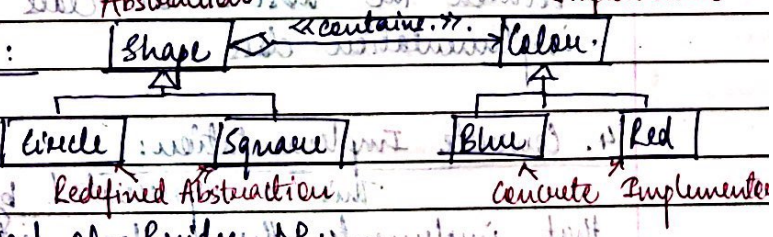
- **Intent:** It is used to decouple an abstraction from its implementation so that the two can vary independently.

↳ Bridge split a large class or a set of closely related classes into two separate hierarchies — abstraction and implementation.

↳ which can be developed independently for each other.

★ ↳ The Original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.

Ex:



Need of Bridge DP:

1. Alternative to inheritance.
2. Isolate Abstraction from implementation.
3. Can be modified independently.
4. Implementation can be changed dynamically at run time.

★ Participants:

1. **Abstraction:**

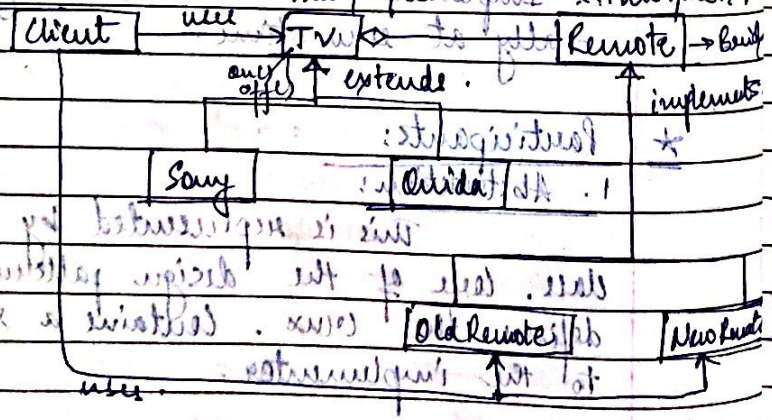
This is represented by an abstract class. Core of the design pattern and defines the API. Contains a reference to the implementor.

2. Refined Abstraction: — This is represented by a class that extends the abstract class.

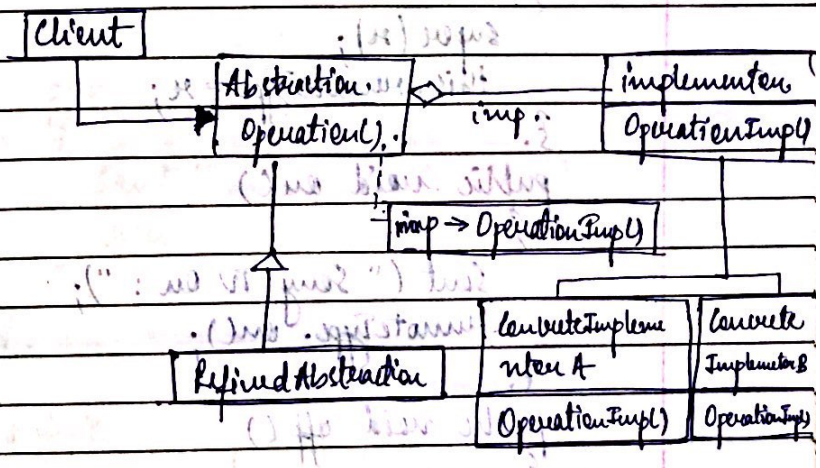
3. Bridge/Implementor: It defines the interface for implementation classes. This is represented by an interface that acts as a bridge between the abstraction class and implementation class.

4. Concrete Implementation: This is represented by a class that implements the bridge interface and provides concrete implementations for the abstract class.

Example:



★ Structure:



Code:

```
abstract class TV {
    Remote remote;
    TV(Remote r)
    {
        this.remote = r;
    }
    abstract void on();
    abstract void off();
}
class Sony extends TV.
{
    Remote remoteType;
```

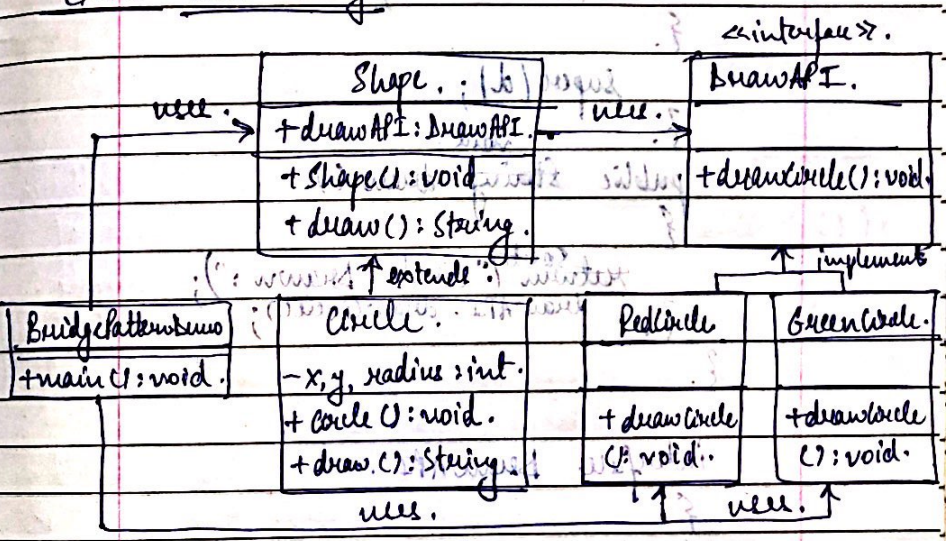
```
Sony (Remote r)
{
    super(r);
    this.remoteType = r;
}
public void on()
{
    Sout("Sony TV on:");
    remoteType.on();
}
public void off()
{
    Sout("Sony Onida TV off:");
    remoteType.off();
}
}
public interface Remote
{
    public void on();
    public void off();
}
class OldRemote implements Remote
{
    public void on()
    {
        Sout("On with old Remote");
    }
    public void off()
    {
        Sout("Off with old Remote");
    }
}
```

```
class NewRemote implements Remote {
    public void on() {
        System.out.println("On with New Remote");
    }
    public void off() {
        System.out.println("Off with New Remote");
    }
}

class Client {
    public static void main(String[] args) {
        TV sonyOldRemote = new Sony(new OldRemote());
        sonyOldRemote.on();
        sonyOldRemote.off();
    }
}
```

O/P:

```
Sony TV is On
On with Old Remote.
Sony TV off:
Off with old Remote.
```

Ex2: Case Study:



Code:

```
abstract class Shape {
    DrawAPI drawAPI;
    Shape(DrawAPI d) {
        this.drawAPI = d;
    }
    abstract void Shape();
    abstract String draw();
}

class Circle extends Shape {
    private int x, y, radius;
```

P.T.O.

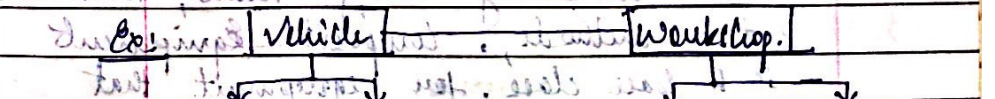
```
Circle(DrawAPI d)
{
    super(d);
}
public void draw()
{
    System.out.println("Circle drawn :");
    drawAPI.drawCircle();
}
interface DrawAPI{
    public void drawCircle();
}
class RedCircle implements DrawAPI.
{
    public void drawCircle();
    {
        System.out.println("Drawn with Red Colour");
    }
}
class GreenCircle implements DrawAPI.
{
    public void drawCircle()
    {
        System.out.println("Drawn with Green Colour");
    }
}
```

```
class BridgePatternDemo
{
    psvm(String a[])
    {
        Shape redCircle = new Circle(new RedCircle());
        redCircle.draw();
        Shape greenCircle = new Circle(new GreenCircle());
        greenCircle.draw();
    }
}
```

O/P:-

Circle Drawn: Drawn with Red Colour.

Circle Drawn: Drawn with Green Colour.



Typical example of Bridge design Pattern.

★ Adiantages:

1. Bridge Pattern decouple an abstraction

from its implementation so that the two can vary independently.

Questions:

1. List any 3 applications of Façade Pattern. Explain Structure and Participants of Façade Pattern with Applicable Example. 4 marks

2. In an application there exists a real subject: "Folder". Contents of folder should be accessible by authenticated users only. Which design pattern is appropriate to implement this authentication process? Implement a Java Program using three Design Patterns. 5 marks

3. Equipment class define an interface for 3 equipments in the part of whole hierarchy using Name() & Price() two methods. Composite Equipment is a base class for equipment that contains other equipments. Implement a Java program using Composite Design Pattern to print payment statement when client must be able to buy any single equipment or composite equipment.

UNIT – 4.

BEHAVIORAL DESIGN PATTERNS – I

Introduction:

1. Behavioral design patterns are concerned with the interaction and responsibility of objects.
2. In these design pattern, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

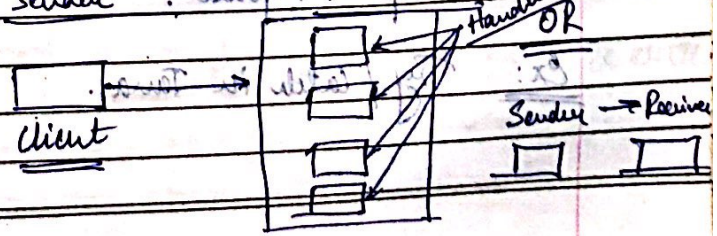
Interaction & Responsibility

Types:

1. Chain of Responsibility Pattern.
2. Command Pattern.
3. Interpreter Pattern.
4. Iterator Pattern.
5. Mediator Pattern.
6. Memento Pattern.
7. Observer Pattern.
8. State Pattern.
9. Strategy Pattern.
10. Template Pattern.
11. Visitor Pattern.
12. Null Object.

Chain of Responsibility: (COR):

Sender . . . Receiver.

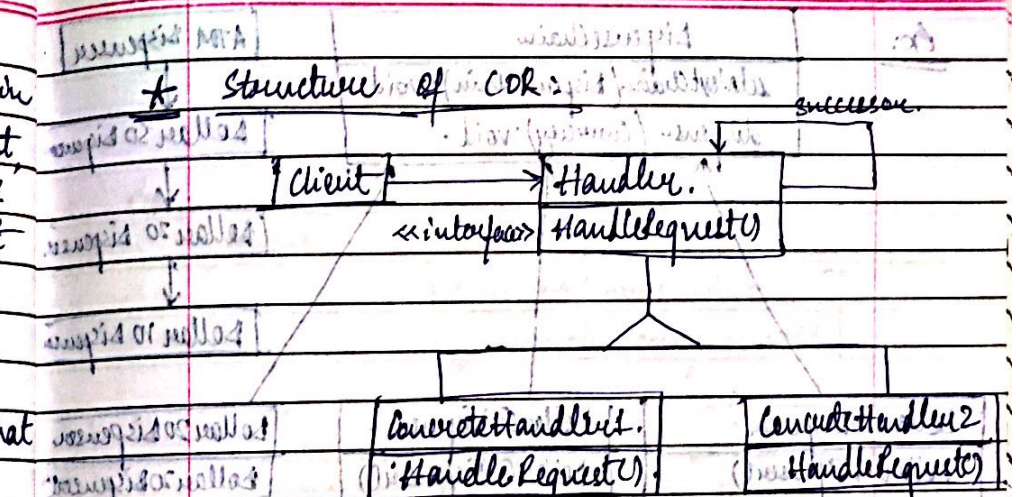


* Client will pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

* INTENT:
  - It is a behavioral design pattern that lets you pass requests along a chain of handlers.
  - Upon receiving a request, each handler decides either to process a request or pass to next handler.
  - The Chain of Responsibility (COR) is used when more than one object handles a request and performs their corresponding responsibility.
  - This pattern can be implemented by using two ways:
    1. One Request Handler will serve the Input Request.
    2. Multiple Request Handlers will serve the Input Request.

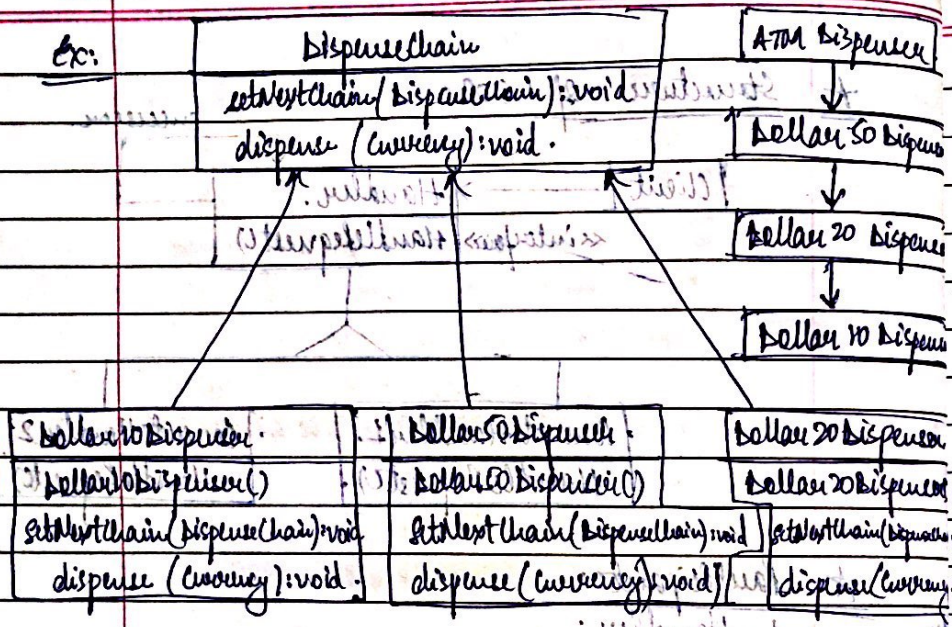
Ex: Try / catch in Java.

* Structure of COR:



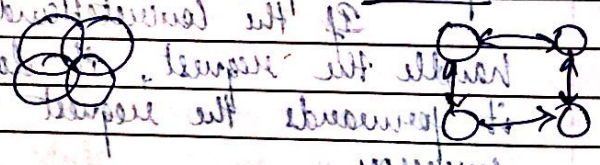
* Participants:
  1. Handler:
     It defines an interface for Handling Requests.
  2. ConcreteHandler:
     Handles Requests. It is responsible for Handlers.
     If the ConcreteHandler can handle the request, it does, otherwise it forwards the request to its successor.
  3. Client:
     Initiates a request to a concrete Handler Object of the chain.

Ex:



* Tight Coupling and Loosely Coupling:

1. More Interdependency
2. More Coordination.
3. More Information flow.
4. Ex:

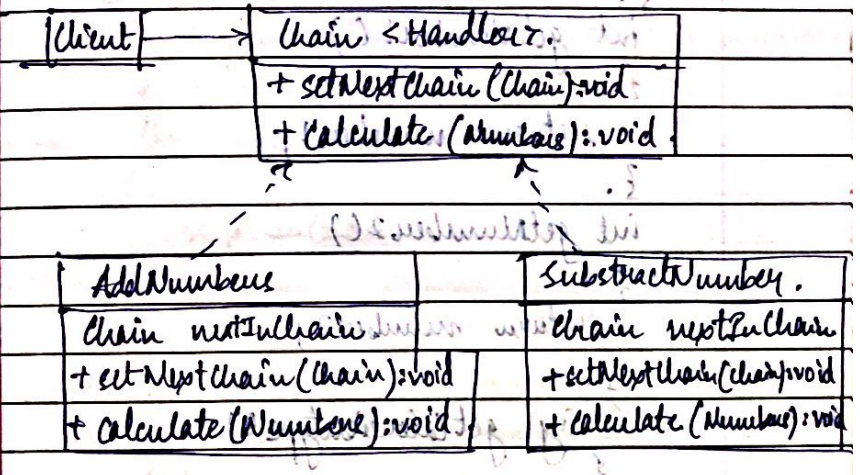


1. Less Interdependency.
2. Less Co-ordination.
3. Less Information flow.
4. Ex:

- Loose Coupling:

Loose coupling is a design concept which aims to reduce the interdependency that is if when a change is made it should not make us do changes to other dependant pro objects..

Ex:



Code:

```
interface Chain
{
    public void setNextChain(Chain nextChain);
    public void calculate(Numbers request);
}

class Numbers
{
    private int number1;
    private int number2;
    private String calculationWanted;
    public Numbers(int n1, int n2, String s)
    {
        number1 = n1;
```

```
number1 = n1;
number2 = n2;
calculation_wanted = s;
}
int getNumber1()
{
return number1;
}
int getNumber2()
{
return number2;
}
String getCalculationType()
{
return calculation_wanted;
}
}
class AddNumbers implements Chain
{
Chain nextInChain;
public void setNextChain(Chain next)
{
nextInChain = next;
}
public void calculate(Numbers request)
{
if (request.getCalculationType() == "add")
{
System.out.println(request.getNumber1() + "+" +
request.getNumber2() + "=" +
(request.getNumber1() + request.getNumber2));
}
else
{
nextInChain.calculate(request);
}
}
class SubNumbers implements Chain
{
Chain nextInChain;
public void setNextChain(Chain next)
{
nextInChain = next;
}
public void calculate(Numbers request)
{
if (request.getCalculationType() == "sub")
{
System.out.println("-----");
}
else
{
System.out.println("Only add and sub are
allowed");
}
}
}
```

class COR.

```
{
    public static void main (String args[])
    {
        Chain c1 = new AddNumbers();
        Chain c2 = new SubNumbers();
        c1.setNextChain(c2);
        Numbers request = new Numbers(4,2,"add");
        c1.calculate(request);
    }
}
```

*** Example: ATM Example:

```
interface DispenseChain
{
    void setNextChain(DispenseChain nextChain);
    void dispense(Currency cur);
}

class Currency
{
    private int amount;
    public Currency(int amt)
    {
        this.amount = amt;
    }
    public getAmount()
    {
        return this.amount;
    }
}

class Dollar50Dispenser implements DispenseChain
{
    private DispenseChain chain;

    public void setNextChain(DispenseChain nextt)
    {
        chain = nextt;
    }
    public void dispense(Currency cur)
    {
        if (cur.getAmount() >= 50)
        {
            int num = cur.getAmount()/50;
            int remainder = cur.getAmount()%50;
            System.out.println("Dispensing "+num+" 50$ note");
            if (remainder != 0)
                this.chain.dispense(new Currency(remainder));
        }
        else
            this.chain.dispense(cur);
    }
}
```

not needed

```
class Dollar20Dispenser implements DispenseChain
{
    ——— same as above class.
}

class Dollar10Dispenser implements DispenseChain
{
    ——— same as above class.
}

public class Client
{
    psvm (s a[])
    {
        DispenseChain c1 = new Dollar50Dispenser();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();
        c1.setNextChain(c2);
        c2.setNextChain(c3);
        while(true)
        {
            Scanner sc=new Scanner(System.in);
            Sout("Enter the amount ");
            int amt = sc.nextInt();
            if(amt%10 != 0)
            {
                Sout("Amount should be in
                    multiples of 10");
                break;
            }
            c1.dispense(amount, new Currency(amount));
        }
    }
}
```

O/P:

Enter the amount - 550.
Dispensing 11 50$ notes.
Enter the amount - 120.
Dispensing 2 50$ notes.
Dispensing 1 20$ notes.
Enter the amount - 345.
Amount should be in multiples of 10.

Interpreter Design Pattern:

Intent: ↳ (1). The pattern defines the grammar of a particular language in an object oriented way which can be evaluated by the interpreter itself.

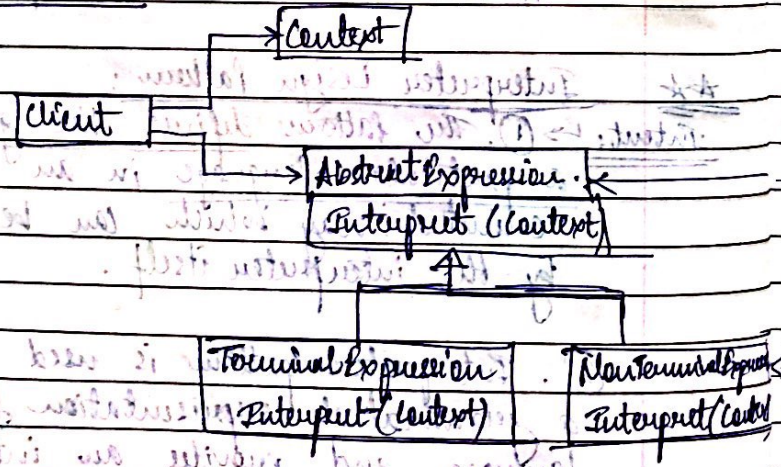
(2). Interpreter pattern is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar.

↳ This pattern involves implementing an expression interface which tells to interpret a particular context.

↳ Applications:
1. SQL parsing. (Google Translator).
2. Symbol Processing Engine, etc.
3. JVM (Ex).

• Participants:
1. Client.
2. Context.
3. Expression. (Abstract)
4. Terminal
5. Non Terminal.

• Structure:



★ • The Interpreter pattern describe how to define a grammar for simple languages. which represent sentences in the language and interpret these sentences.

★ Iterator Design Pattern:
1. Collection — List — container (group of objects)
2. Iterator. — sequential.
↳ hasNext()
↳ next()
↳ remove().

↳ One of the Behavioral Pattern.
↳ Iterator pattern is used to provide a standard way to traverse through a group of objects.
↳ Aggregation between collection and Iterator.
↳ Not just traversal but can be done in different kind of iterators based on client requirements.

• Intent:
1. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
2. Iterator is a behavioral pattern that lets you traverse elements of a collection without its underlying

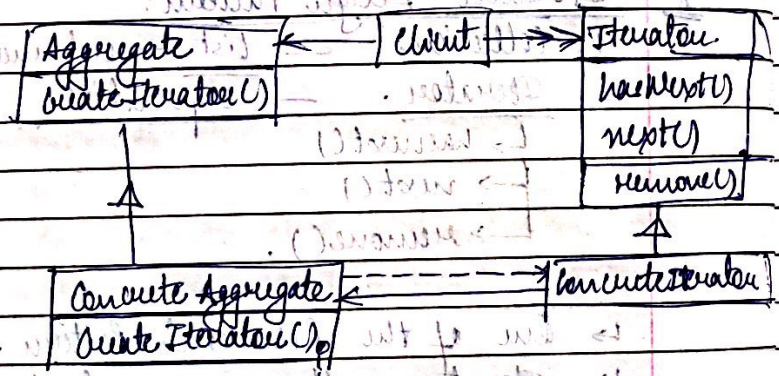
Case Study: Decorator, AbstractFactory, Composite, Bridge.

Page No.
Date :

implementation (list, stack, tree, etc).

3. It is also known as Cursor.

• Structure:



• Participants:

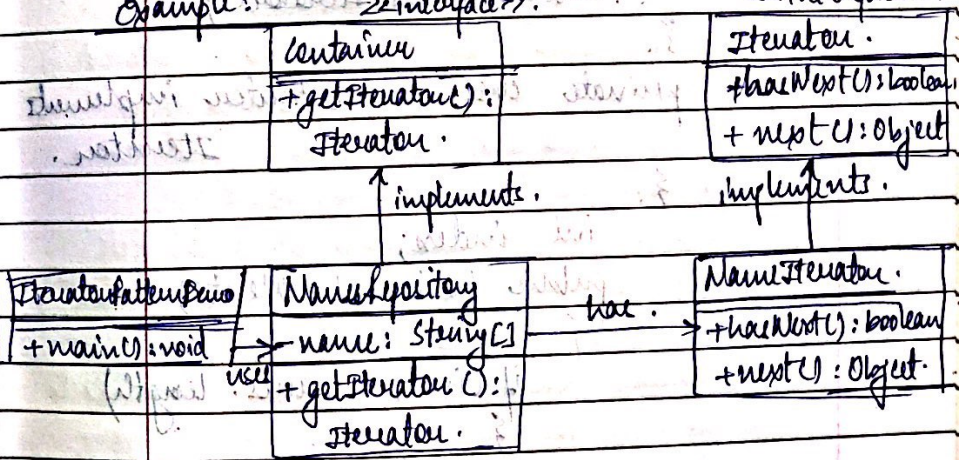
1. Iterator:
↳ Define an interface for accessing and traversing elements.

2. ConcreteIterator:
↳ Implements the Iterator interface.
↳ Keep tracks of the current position in the aggregate traversal of the aggregate

3. Aggregate:
↳ Define an interface for creating an Iterator Object.

4. ConcreteAggregate:
↳ Implements the creation iterator creation interface to return an instance of the proper concrete iterator.

Example:



Code:

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

```
public class NameRepository implements Container.
{
    public String names[] = {"Robert", "James", "Julie", "John"};
    public Iterator getIterator()
    {
        return new NameIterator();
    }.
    private class NameIterator implements Iterator.
    {.
        int index;
        public boolean hasNext()
        {
            if (index < names.length)
            {
                return true;
            }
            return false;
        }.
        public Object next()
        {
            if (this.hasNext())
            {
                return names[index++];
            }
            return null;
        }.
    } // private class NameIterator end.
} // public class NameRepository end.

class Client.
{
    psvm (s a[])
    {.
        NameRepository nr = new NameRepository();
        for (Iterator iter = nr.getIterator();
             iter.hasNext();)
        {
            String name = (String) iter.next();
            Sout ("Name :" + name);
        }
    }.
}
```

O/P:

```
Name : Robert.
Name : James.
Name : Julie.
Name : John.
```

UNIT – 5.

BEHAVIOUR DESIGN PATTERNS – II.

- Patterns:
  - ★ 1. Mediator
  - 2. Memento.
  - ★ 3. Observer
  - 4. State
  - 5. Strategy.
  - ★ 6. Template

1. Mediator Design Pattern:

↳ Mediator design pattern is used to provide a centralized communication medium between different objects in a system.

↳ deals with the behaviour of objects.

Intent: ↳ 1. Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other.

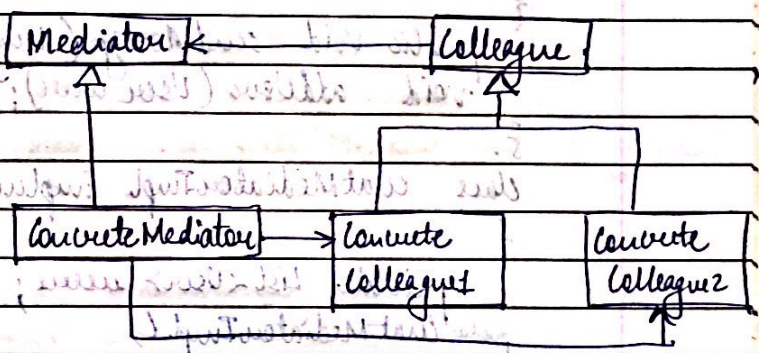
2. ↳ Allows for the action of each object set to vary independently of each other.

↳ Used in enterprise applications where multiple objects are interacting with each other.

★ ↳ Provides a mediator between objects for communication and help in implementing loose coupling between objects.

↳ Ex: Facebook, Air Traffic Controller (ATC), Router.

- Structure:



- Participants:

1. Mediator: It define the interface for communication between colleague objects.

2. ConcreteMediator: It implements the mediator interface and co-ordinates between Colleague objects.

3. Colleague: It defines the interface for communication with other colleagues.

4. ConcreteColleague: It implements the colleague interface and define concrete implementation for the colleagues and communicates with other colleagues through its mediator.

P.T.O.

• Example: Chat Application - from PPT:

```
interface ChatMediator
{
    public void sendMessage(String msg, User user);
    void addUser(User user);
}
class ChatMediatorImpl implements ChatMediator
{
    private List<User> users;
    public ChatMediatorImpl()
    {
        this.users = new ArrayList<>();
    }
    public void addUser(User user)
    {
        this.users.add(user);
    }
    public void sendMessage(String msg, User
    {
        for(User u : this.users)
        {
            // message should not be received by the sender
            if(u != user)
            {
                u.receive(msg);
            }
        }
    }
```

```
abstract class User
{
    protected ChatMediator mediator;
    protected String name;
    public User(ChatMediator med, String name)
    {
        this.mediator = med;
        this.name = name;
    }
    public abstract void send(String msg);
    public abstract void receive(String msg);
}
class UserImpl extends User
{
    public UserImpl(ChatMediator med, String name)
    {
        super(med, name);
    }
    public void send(String msg)
    {
        System.out("this.name + "Sending Message" + msg);
        mediator.sendMessage(msg, this);
    }
    public void receive(String msg)
    {
        System.out("this.name + "Received Message" + msg);
    }
}
```

```
class Client
{
    psvm(SA[])
    {
        ChatMediator med = new ChatMediatorImpl();
        User user1 = new UserImpl(med, "A");
        User user2 = new UserImpl(med, "B");
        User user3 = new UserImpl(med, "C");
        User user4 = new UserImpl(med, "D");
        med.addUser(user1);
        med.addUser(user2);
        med.addUser(user3);
        med.addUser(user4);
        user1.send("Hi...");
        user4.send("Hello...");
    }
}
```

O/P:

A: sending Message: Hi...
B: Received Message: Hi...
C: Received Message: Hi...
D: Received Message: Hi...

Example 2: Facebook Chat Server

same as above example.

Example: Air Traffic Controller:

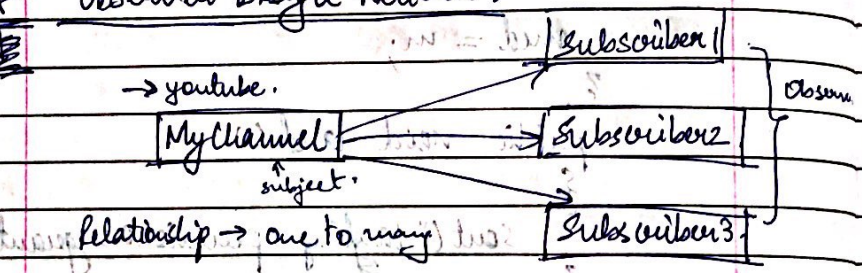
```
interface Mediator
{
    public void registerRunway(Runway runway);
    public void registerFlight(Flight flight);
    public boolean isLandingOK();
}
class MediatorImpl implements Mediator
{
    private Flight flight;
    private Runway runway;
    public boolean land;
    public void registerRunway(Runway runway)
    {
        this.runway = runway;
    }
    public void registerFlight(Flight flight)
    {
        this.flight = flight;
    }
    public boolean isLandingOK()
    {
        return land;
    }
}
interface Command
{
    void land();
}
```

```
class Flight implements Command
{
    private Mediator med;
    public Flight(Mediator m)
    {
        med=m;
    }
    public void land()
    {
        if(med.isLandingOk())
        {
            Sout("Succesfully Landed.");
        }
        else
        {
            Sout("Waiting to Land");
        }
    }
    public void getReady()
    {
        Sout("Ready to Land");
    }
}

class Runway implements Command
{
    private Mediator med;
    public Runway(Mediator m)
    {
        med = m;
    }
    public void land()
    {
        Sout("Landing permission granted");
    }
}

public class Med_AirPlane
{
    psvm(sArI)
    {
        Mediator atc = new MediatorImpl();
        Flight fl = new Flight(atc);
        Runway main = new Runway(atc);
        atc.registerFlight(fl);
        atc.registerRunway(main);
        fl.getReady();
        main.land();
        fl.land();
    }
}
```

O/P:

Ready to Land.
Landing Permission granted.
Succesfully Landed.

★★ Observer Design Pattern:



- Known as Publish/Subscriber, Dependance.
  Relationship → one to many.

Uses MVC (Model View Controller).
jsp server.

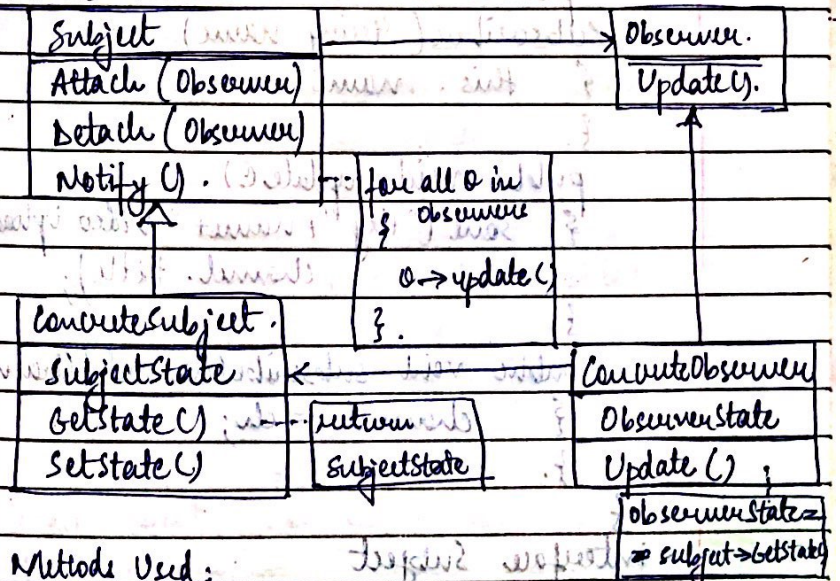
Points to remember:
- Publish – Subscribe = Alternate Name.
- Dependents = Alternate Name.
- one to many mapping.
- Notify } Methods
- Update }

- Observer pattern is useful when you are interested in the state of an object and want to get notified whenever there is change.
- The object that watch the state of another object are called Observer & the object that is being watched is called Subject.
- Ex: Bidding System.

- Intent:
  Define a one to many dependancy between objects so that when one object changes state, all its dependants are notified and updated automatically.

- Subject contains a list of observers to notify of any change in its state.
- MVC uses Observer pattern.
  ↳ Model is the subject.
  ↳ Views are the observers.

- Structure:



- Methods Used:
  ↳ Subscribe(), Unsubscribe(), Notify(), Update().

- Participants:

1. Subject – interface/abs class defines, attach & detach methods
2. ConcreteSubject -
3. Observer – interface / abs class defines operation for notify
4. ConcreteObserver -

- Code:

```
interface Observer {
    void subscribeChannel(Channel ch);
    void update();
}
class Subscriber implements Observer
{   private String name;
    private Channel channel = new Channel();
    Subscriber(String name)
    {   this.name = name;
    }
    public void update()
    {   sout("Hey "+name+" Video Uploaded "+
            channel.title);
    }
    public void subscribeChannel(Channel ch)
    {   channel = ch;
    }
}
interface Subject
{   void subscribe(Subscriber sub);
    void unsubscribe(Subscriber sub);   // register / unregister observers.
    void notifySubscribers();
    void upload(String title);
}
class Channel implements Subject
{
    List<Subscriber> subs = new ArrayList<>();
    public String title;
    public void subscribe(Subscriber sub)
    {   subs.add(sub);
    }
    public void unsubscribe(Subscriber sub)
    {   subs.remove(sub);
    }
    public void notifySubscribers()
    {   for(Subscriber sub : subs)
        {   sub.update();
        }
    }
    public void upload(String title)
    {   this.title = title;
        notifySubscribers();
    }
}
class Client
{
    psvm(S a[])
```

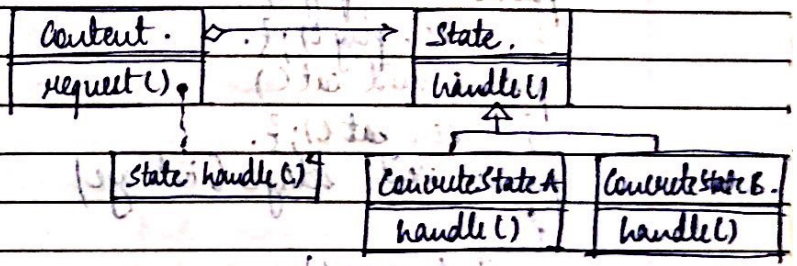
```
{
    Channel java = new Channel();
    Subscriber s1 = new Subscriber("John");
    ——— s2 = ——— ("David");
    ——— s3 = ——— ("Mike");
    ——— s4 = ——— ("Harvey");
    java.subscribe(s1);
    java.subscribe(s2);
    java.subscribe(s3);
    java.subscribe(s4);
    java.unsubscribe(s3);
    Sout("Subscribing Channel");
    s1.subscribeChannel(java);
}
```

★★★ STATE DESIGN PATTERN:

- **Intent:** 1. Allow an object to alter its behavior when its internal state changes.
  2. The object will appear to change its class.
- **Also Known As:** Objects for States.
- Context forwards the request to the state object for processing.

Ex: TV Remote with a single button to switch ON or OFF with it.

- **Structure:**



- **Participants:**

1. **State** — The interface defines operations which each state must handle.
2. **Concrete State:** — The classes which contain the state specific behaviour.
3. **Context:** —

Ex: Kid Example.

```
interface kidstate          // State.
{   public void play();
    public void eat();
}.
class Kid                   // Context.
{
    private int age;
    private kidstate ks;
    Kid(int age)
    { this.setage(age); }  this.stage(age);
```

```
public void play()
{ ks.play(); }
public void eat()
{ ks.eat(); }
public void setAge(int age)
{
    if (age==1)
    { ks = new FirstYearKid(); }
    else if (age==2)
    { ks = new SecondYearKid(); }
    else if (age==3)
    { ks = new ThirdYearKid(); }
    else if (age==4)
    { ks = new FourthYearKid(); }
    else
        ks = new FirstYearKid();
}
}
class FirstYearKid implements KidState.
{
    public void play()
    { System.out.println("Play with cradle"); }
    public void eat()
    { System.out.println("Drink Application"); }
}
// same for 2nd, 3rd, 4th year kid.
```

```
class Client
{
    psvm(String a[])
    {   Kid kid = new Kid(2);
        System.out.println("2nd year kid");
        kid.eat();
        kid.play();
        System.out.println("4th year kid");
        kid.setAge(4);
        kid.eat();
        kid.play();
    }
}
```

Example and Strategy design Pattern in pics.

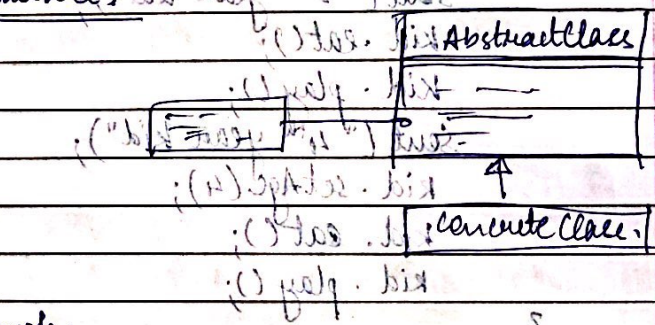
TEMPLATE Design Pattern:

↳ It is concerned of states and steps.
↳ Preset structures.
↳ used in framework development.

Ex: To build a house.

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Base class declare algorithm placeholders and derived classes implement the placeholders.



- Structural:
- Example:

```
abstract class HouseTemplate
{
    public final void buildHouse()
    {
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built");
    }
    private void buildWindows()
    {
        System.out.println("Building windows");
    }
    private void buildFoundation()
    {
        System.out.println("Building Foundation");
    }
    public abstract void buildPillars();
    public abstract void buildWalls();
}
class House extends HouseTemplate
{
    public void buildPillars()
    {
        System.out.println("Building Pillars");
    }
    public void buildWalls()
    {
        System.out.println("Building walls");
    }
}
class Client
{
    public static void main(String[] args)
    {
        HouseTemplate ht = new House();
        ht.buildHouse();
    }
}
```

O/P:

```
Building Foundation.
Building Pillars.
Building walls.
Building windows.
House is Built.
```

Memento Design Pattern: (Memorization)

↳ Known as: Snapshot, Token.

↳ It deals with the previous state of an object without revealing details about implementation.

↳ Implemented using two objects:

1. Originator.
2. CareTaker.

- Intent: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to its state later.

- Structure: