Design Patterns

Design Patterns in Java

Notes By Adipta Basu

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Advantage of design pattern:

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don’t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

▼ Core Java Design Patterns

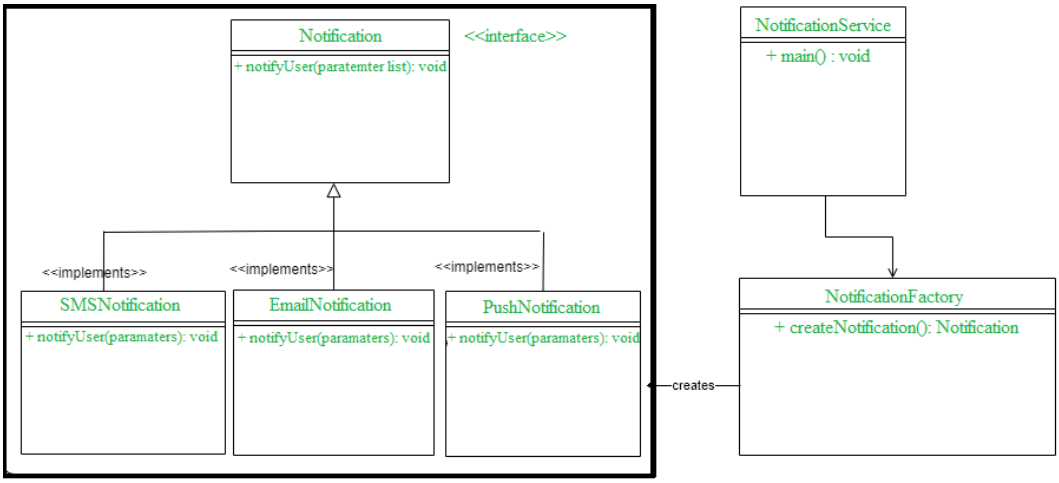
▼ Creational Design Pattern

▼ Factory Pattern

It is a creational design pattern which talks about the creation of an object. The factory design pattern says that define an interface (A java interface or an abstract class) and let the subclasses decide which object to instantiate. The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses. Since this design patterns talk about instantiation of an object and so it comes under the category of creational design pattern. If we notice the name Factory method, that means there is a method which is a factory, and in general factories are involved with creational stuff and here with this an object is being created. It is one of the best ways to create an object where object creation logic is hidden to the client. Now Let's look at the implementation.

▼ Implementation

1. Define a factory method inside an interface.
2. Let the subclass implements the above factory method and decide which object to create.
   In Java constructors are not polymorphic, but by allowing subclass to create an object, we are adding polymorphic behavior to the instantiation. In short, we are trying to achieve Pseudo polymorphism by letting the subclass to decide what to create, and so this Factory method is also called as Virtual constructor.



▼ Step 1 Create Notification Interface

```
public interface Notification
{
    void notifyUser();
}
```

▼ Step 2 Create the Classes that implement the Notification Interface

Creating SMSNotification Class

```
public class SMSNotification implements Notificatio
{
    @Override
    public void notifyUser()
    {
        System.out.println("Sending a SMS Notificat
    }
}
```

Creating EmailNotification Class

```
public class EmailNotification implements Notificat
{
```

```
    @Override
    public void notifyUser()
    {
        System.out.println("Sending a Email Notific
    }
}
```

Creating PushNotification Class

```
public class PushNotification implements Notificati
{
    @Override
    public void notifyUser()
    {
        System.out.println("Sending a Push Notifica
    }
}
```

▼ Step 3 Create the Factory Class to instantiate a concrete class

Creating NotificationFactory Class

```
public class NotificationFactory
{
    public Notification createNotification(String c
    {
        if(channel==null || channel.isEmpty())
        {
            return null;
        }
        else
        {
            switch (channel)
            {
        case "SMS":
            return new SMSNotification();
```

```
            case "EMAIL":
                return new EmailNotification();
            case "PUSH":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unk
           }
            }
    }
}
```

▼ Step 4 Use the Factory Class to get the Object of Concrete Class

Creating notificationService class which uses NotificationFactory Class

```
public class NotificationService
{
    public static void main(String[] args)
    {
        NotificationFactory notificationFactory = n
        Notification notification = notificationFac
        notification.notifyUser();
    }
}
```

💡 `Output :` `Sending an SMS notification`

▼ Real-Time Examples

This design pattern has been widely used in JDK, such as

1. getInstance() method of java.util.Calendar, NumberFormat, and ResourceBundle uses factory method design pattern.
2. All the wrapper classes like Integer, Boolean etc, in Java uses this pattern to evaluate the values using valueOf() method.

3. java.nio.charset.Charset.forName(), java.sql.DriverManager#getConnection(), java.net.URL.openConnection(), java.lang.Class.newInstance(), java.lang.Class.forName() are some of ther example where factory method design pattern has been used.

▼ Abstract Factory Pattern

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

▼ Implementation

Basically, it like the Factory Pattern, but instead of having a singular producer for the concrete classes, we create a factory of factories, that create the concrete classes.

▼ Step 1 Create the Enums

Creating the Colors Enum

```
package color;

public enum Colors
{
    RED,BLUE
}
```

Creating the Shapes Enum

```
package shape;

public enum Shapes
{
```

```
    RECTANGLE,CIRCLE
}
```

▼ Step 2 Create the Interfaces

Create the Shape Interface

```
package shape;

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

Create the Color Interface

```
package color;

public interface Color
{
    public void fill();
}
```

▼ Step 3 Create the Concrete Classes

Creating the Red Class

```
package color;

public class Red implements Color
{
    @Override
    public void fill()
    {
        System.out.println("In the fill method for
```

```
    }
}
```

Creating the Blue Class

```
package color;

public class Blue implements Color
{
    @Override
    public void fill()
    {
        System.out.println("In the fill method for
    }
}
```

Creating the Circle Class

```
package shape;

public class Circle implements Shape
{
    @Override
    public void draw()
    {
        System.out.println("Inside the draw method
    }
}
```

Creating the Rectangle Class

```
package shape;

public class Rectangle implements Shape
{
    @Override
```

```
    public void draw()
    {
        System.out.println("Inside the draw method
    }
}
```

▼ Step 4 Create the Abstract Factory for the Factory of Factories

Create the AbstractFactory Class

```
package factory;

import color.Color;
import color.Colors;
import shape.Shape;
import shape.Shapes;

public abstract class AbstractFactory
{
    abstract Color getColor(Colors color);
    abstract Shape getShape(Shapes shape);
}
```

▼ Step 5 Create the Factories extending the Abstract Factory

Creating the ColorFactory Class

```
package factory;

import color.Blue;
import color.Color;
import color.Colors;
import color.Red;
import shape.Shape;
import shape.Shapes;

public class ColorFactory extends AbstractFactory
```

```
{
    @Override
    Color getColor(Colors color)
    {
        if(color==null)
        {
            return null;
        }
        switch (color)
        {
            case BLUE:
            {
                return new Blue();
            }
            case RED:
            {
                return new Red();
            }
            default:
                throw new IllegalArgumentException(
        }
    }
    @Override
    Shape getShape(Shapes shape)
    {
        return null;
    }
}
```

Creating the ShapeFactory Class

```
package factory;

import color.Color;
import color.Colors;
import shape.Circle;
```

```
import shape.Rectangle;
import shape.Shape;
import shape.Shapes;

public class ShapeFactory extends AbstractFactory
{
    @Override
    Color getColor(Colors color)
    {
        return null;
    }
    @Override
    Shape getShape(Shapes shape)
    {
        if(shape==null)
        {
            return null;
        }
        switch (shape)
        {
            case CIRCLE:
            {
                return new Circle();
            }
            case RECTANGLE:
            {
                return new Rectangle();
            }
            default:
                throw new IllegalArgumentException(
        }
    }
}
```

▼ Step 6 Creating the Factory Producer, i.e., a Factory of Factories

Creating a FactoryProducer

```
package factory;

public class FactoryProducer
{
    public static AbstractFactory getFactory(String
    {
        if("SHAPE".equalsIgnoreCase(choice))
        {
            return new ShapeFactory();
        }
        else if("COLOR".equalsIgnoreCase(choice))
        {
            return new ColorFactory();
        }
        return null;
    }
}
```

▼ Step 7 Create a class that invokes the Factory Producer

Creating the AbstractFactoryPatternDemo Class

```
package factory;

import color.Color;
import color.Colors;
import shape.Shape;
import shape.Shapes;

public class AbstractFactoryPatternDemo
{
    public static void main(String args[])
    {
        AbstractFactory shapeFactory=FactoryProduce
```

```
        Shape shape1=shapeFactory.getShape(Shapes.C
        shape1.draw();

        Shape shape2=shapeFactory.getShape(Shapes.R
        shape2.draw();


        AbstractFactory colorFactory=FactoryProduce

        Color color1=colorFactory.getColor(Colors.B
        color1.fill();

        Color color2=colorFactory.getColor(Colors.R
        color2.fill();
    }
}
```

Output:

```
💡 Inside the draw method of the Circle Class
   Inside the draw method of the Rectangle Class
   In the fill method for the Blue Class
   In the fill method for the Red Class
```

▼ Singleton Pattern

The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Definition: The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Implementation

Method 1 Classical Implementation

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.

```
// Classical Java implementation of singleton
// design pattern
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

Method 2 Making getInstance() synchronised

Here using synchronised makes sure that only one thread at a time can execute getInstance().

The main disadvantage of this is method is that using synchronised every time while creating the singleton object is expensive and may decrease the performance of your program.

However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstanc
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

▼ Method 3 Eager Loading

Here we have created instance of singleton in static initialiser. JVM executes static initialiser when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj = new Singleton();

    private Singleton() {}
```

```
    public static Singleton getInstance()
    {
        return obj;
    }
}
```

▼ Method 4 Double Checked Locking(Best)

If you notice carefully once an object is created synchronisation is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.

So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronise the first way through, just what we want.

We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialised to Singleton instance. This method drastically reduces the overhead of calling the synchronised method every time.

```
class Singleton
{
    private static volatile Singleton obj  = null;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
```

```
                    obj = new Singleton();
                }
            }
            return obj;
        }
    }
```

▼ Prototype Pattern

Prototype 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 same properties only if required. This approach saves costly resources and time, especially when object creation is a heavy process.
The prototype pattern is a creational design pattern. Prototype patterns are required, when object creation is time consuming, and costly operation, so we create objects with the existing object itself. One of the best available ways to create an object from existing objects is the clone() method. Clone is the simplest approach to implement a prototype pattern. However, it is your call to decide how to copy existing object based on your business model.

▼ Prototype Design Participants

1. Prototype : This is the prototype of an actual object.
2. Prototype registry : This is used as a registry service to have all prototypes accessible using simple string parameters.
3. Client : Client will be responsible for using registry service to access prototype instances.

▼ Implementation

▼ Step 1 Create the Abstract Class that Concrete Classes will Implement.

Creating the Color Class

```
package color;

public abstract class Color implements Cloneable
{
    protected String colorName;

    abstract void addColor();

    public Object clone()
    {
        Object clone=null;
        try
        {
            clone=super.clone();
        } catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
        return clone;
    }
}
```

▼ Step 2 Create the Concrete Classes

Creating the BlackColor Class

```
package color;

public class BlackColor extends Color
{
    public BlackColor()
    {
        this.colorName="black";
    }
```

```
    @Override
    void addColor()
    {
        System.out.println("Black Color Added");
    }
}
```

Creating the BlueColor Class

```
package color;

public class BlueColor extends Color
{
    public BlueColor()
    {
        this.colorName="blue";
    }

    @Override
    void addColor()
    {
        System.out.println("Blue Color Added");
    }
}
```

▼ Step 3 Create the Class that Stores the Objects

Creating the ColorStore Class

```
package color;

import java.util.HashMap;
import java.util.Map;

public class ColorStore
{
```

```
    private static Map<String, Color> colorMap=new

    static
    {
        colorMap.put("blue", new BlueColor());
        colorMap.put("black", new BlackColor());
    }

    public static Color getColor(String colorName)
    {
        return (Color)colorMap.get(colorName).clone
    }
}
```

▼ Step 4 Call the Object Store that was created

Creating the main Runner Class

```
package color;

public class PrototypeDesignPattern
{
    public static void main(String[] args)
    {
        ColorStore.getColor("blue").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("blue").addColor();
    }
}
```

Output:

Blue Color Added
Black Color Added
Black Color Added
Blue Color Added

▼ Advantages of Prototype Design Pattern

- Adding and removing products at run-time – Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
- Specifying new objects by varying values – Highly dynamic systems let you define new behavior through object composition by specifying values for an object's variables and not by defining new classes.
- Specifying new objects by varying structure – Many applications build objects from parts and subparts. For convenience, such applications often let you instantiate complex, user-defined structures to use a specific subcircuit again and again.
- Reduced subclassing – Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.

▼ Builder Pattern

Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations." It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

▼ Implementation

▼ Step 1 Create a concrete class

Creating the Product Class

```
package product;

public class Product
{
    private String productName;
    private float price;
    private String owner;

    public Product(String productName, float price,
    {
        this.productName = productName;
        this.price = price;
        this.owner = owner;
    }

    public Product()
    {
    }

    public String getProductName()
    {
        return productName;
    }

    public void setProductName(String productName)
    {
        this.productName = productName;
    }

    public float getPrice()
    {
        return price;
```

```
    }

    public void setPrice(float price)
    {
        this.price = price;
    }

    public String getOwner()
    {
        return owner;
    }

    public void setOwner(String owner)
    {
        this.owner = owner;
    }

    @Override
    public String toString()
    {
        return "Product [productName=" + productNam
    }
}
```

▼ Step 2 Create the builder for concrete class

Creating the ProductBuilder Class

```
package product;

public class ProductBuilder
{
    private String productName;
    private float price;
    private String owner;
```

```
    public ProductBuilder()
    {
    }

    public ProductBuilder setProductName(String pro
    {
        this.productName = productName;
        return this;
    }
    public ProductBuilder setPrice(float price)
    {
        this.price = price;
        return this;
    }
    public ProductBuilder setOwner(String owner)
    {
        this.owner = owner;
        return this;
    }
    public Product build()
    {
        return new Product(productName, price, owne
    }
}
```

▼ Step 3 Create the runner class to invoke the builder class

Creating the BuilderDesignPattern Class

```
package product;

public class BuilderDesignPattern
{
    public static void main(String[] args)
    {
        Product p = new ProductBuilder()
```

```
                            .setPrice(100f)
                            .setProductName("Test")
                            .setOwner("Me")
                            .build();

        System.out.println(p);
    }
}
```

▼ Advantages of Builder Design Pattern

1. The parameters to the constructor are reduced and are provided in highly readable method calls.
2. Builder design pattern also helps in minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor.
3. Object is always instantiated in a complete state.
4. Immutable objects can be built without much complex logic in the object building process.

▼ Structural Design Pattern

▼ Adapter Pattern

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

▼ Implementation

▼ Step 1. Create the Interface and Concrete Classes that we use.

Creating the Bird Interface

```
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interfac
    public void fly();
```

```
    public void makeSound();
}
```

Creating the concrete class that implements the above interface.

```
class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
```

▼ Step 2. Being familiar with the Concrete Call and/or the interface.

Creating the ToyDuck Interface

```
interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}
```

Creating the PlasticToyDuck Concrete Class that implements the ToyDuck Interface

```
class PlasticToyDuck implements ToyDuck
{
    public void squeak()
```

```
    {
        System.out.println("Squeak");
    }
}
```

▼ Step 3. Writing an Adapter Class.

Implementing the BirdAdapter Class, where we implement the Interface, that the client implements, so that we can adapt out Object accordingly.

```
class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}
```

▼ Step 4. Invoking the Adapter Class properly.

In the Runner Class, we basically use the BirdAdapter the cast our object to the client acceptable object.

```
class Main
{
```

```
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(spar

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

Output:

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
Chirp Chirp
```

▼ Advantages

1. Helps achieve reusability and flexibility.

2. Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

▼ Bridge Pattern

The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern.

▼ Implementation

▼ Step 1. Create the Core Abstraction, that contains the reference to the implementors.

Creating the Vehicle Abstract Class

```
// abstraction in bridge pattern
abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;

    protected Vehicle(Workshop workShop1, Workshop
    {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();
}
```

▼ Step 2. Extend the abstraction, take the finer details one step below. But hide them from the implementors.

Creating the Car Class

```
// Refine abstraction 1 in bridge pattern
class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workSho
    {
        super(workShop1, workShop2);
```

```
    }

    @Override
    public void manufacture()
    {
        System.out.print("Car ");
        workShop1.work();
        workShop2.work();
    }
}
```

Creating the Bike Class

```
// Refine abstraction 2 in bridge pattern
class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workSh
    {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture()
    {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }
}
```

▼ Step 3. Define the interface for the implementation classes.

Creating the Workshop Interface

```
// Implementor for bridge pattern
interface Workshop
{
```

```
    abstract public void work();
}
```

▼ Step 4. Implements the Implementer by providing the concrete implementations.

Creating the Produce Class

```
// Concrete implementation 1 for bridge pattern
class Produce implements Workshop {
    @Override
    public void work()
    {
        System.out.print("Produced");
    }
}
```

Creating the Assemble Class

```
// Concrete implementation 2 for bridge pattern
class Assemble implements Workshop {
    @Override
    public void work()
    {
        System.out.print(" And");
        System.out.println(" Assembled.");
    }
}
```

▼ Step 5. Create A Runner Class to check the Implementation.

Creating the BridgePattern Class to check the implementation.

```
// Demonstration of bridge design pattern
class BridgePattern {
    public static void main(String[] args)
    {
```

```
        Vehicle vehicle1 = new Car(new Produce(), r
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(),
        vehicle2.manufacture();
    }
}
```

Output:

```
Car Produced And Assembled.
Bike Produced And Assembled.
```

▼ Advantages

1. Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
2. It is used mainly for implementing platform independence features.
3. It adds one more method level redirection to achieve the objective.
4. Publish abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
5. Use bridge pattern to run-time binding of the implementation.
6. Use bridge pattern to map orthogonal class hierarchies.
7. Bridge is designed up-front to let the abstraction and the implementation vary independently.

▼ Composite Pattern

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

▼ Implementation

▼ Step 1. Create the Component, where the Component declares the interface for objects in the composition and for accessing and managing its child components.

Create the Employee Interface

```
public interface Employee
{
    public void showEmployeeDetails();
}
```

▼ Step 2. Create the Leaf, Leaf defines the behaviour for the objects in the composition. It represents leaf objects in the composition.

Create the Developer Leaf Class

```
public class Developer implements Employee
{
        private String name;
    private long empId;
    private String position;

    public Developer(long empId, String name, Strin
    {
        this.empId = empId;
        this.name = name;
        this.position = position;
    }
    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+" "+posi
    }
}
```

Create the Manager Leaf Class

```
public class Manager implements Employee
{
        private String name;
    private long empId;
    private String position;

    public Manager(long empId, String name, String
    {
        this.empId = empId;
        this.name = name;
        this.position = position;
    }
    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+" "+posi
    }
}
```

▼ Step 3. Create the Composite, the Composite stores the child components and implements child related operations in the Component Interface.

Create the CompanyDirectory Class, which stores the Leaf Classes

```
import java.util.ArrayList;
import java.util.List;

public class CompanyDirectory implements Employee
{
    private List<Employee> employeeList = new Array

    @Override
```

```
    public void showEmployeeDetails()
    {
        for(Employee emp:employeeList)
        {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(Employee emp)
    {
        employeeList.add(emp);
    }

    public void removeEmployee(Employee emp)
    {
        employeeList.remove(emp);
    }
}
```

▼ Step 4. Create the Client, which manipulates the objects in the composition through the component interface.

Creating the Component Class

```
public class Company
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer(100, "Lokesh
        Developer dev2 = new Developer(101, "Vinay
        CompanyDirectory engDirectory = new Company
        engDirectory.addEmployee(dev1);
        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra C
        Manager man2 = new Manager(201, "Vikram Sha
```

```
        CompanyDirectory accDirectory = new Company
        accDirectory.addEmployee(man1);
        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDir
        directory.addEmployee(engDirectory);
        directory.addEmployee(accDirectory);
        directory.showEmployeeDetails();
    }
}
```

Output:

```
100 Lokesh Sharma Pro Developer
101 Vinay Sharma Developer
200 Kushagra Garg SEO Manager
201 Vikram Sharma  Kushagra's Manager
```

▼ When to use Composite Design Pattern

Composite Pattern should be used when clients need to ignore the difference between compositions of objects and individual objects. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice, it is less complex in this situation to treat primitives and composites as homogeneous.

1. Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
2. 1. Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

▼ When not to use Composite Design Pattern

1. Composite Design Pattern makes it harder to restrict the type of components of a composite. So it should not be used when you don’t want to represent a full or partial hierarchy of objects.
2. Composite Design Pattern can make the design overly general. It makes harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can’t rely on the type system to enforce those constraints for you. Instead you’ll have to use run-time checks.

▼ Decorator Pattern

Decorator design pattern allows us to dynamically add functionality and behaviour to an object without affecting the behaviour of other existing objects within the same class. We use inheritance to extend the behaviour of the class. This takes place at compile-time, and all the instances of that class get the extended behaviour.

▼ Implementation

▼ Step 1. Create an Interface.

Creating the Shape Interface

```
// Interface named Shape
public interface Shape
{
    // Method inside interface
    void draw();
}
```

▼ Step 2. Create Concrete Classes implementing the same interface.

Creating the Rectangle Concrete Class

```
// Class 1
// Class 1 will be implementing the Shape interface
```

```
// Rectangle.java
public class Rectangle implements Shape
{
    // Overriding the method
    @Override public void draw()
    {
        // /Print statement to execute when
        // draw() method of this class is called
        // later on in the main() method
        System.out.println("Shape: Rectangle");
    }
}
```

Creating the Circle Concrete Class

```
// Circle.java
public class Circle implements Shape
{
    @Override
    public void draw()
    {
        System.out.println("Shape: Circle");
    }
}
```

▼ Step 3. Create an abstract decorator class implementing the above same interface.

Creating the ShapeDecorator Class

```
// Class 2
// Abstract class
// ShapeDecorator.java
public abstract class ShapeDecorator implements Sha
{
    // Protected variable
```

```
    protected Shape decoratedShape;

    // Method 1
    // Abstract class method
    public ShapeDecorator(Shape decoratedShape)
    {
        // This keywordd refers to current object i
        this.decoratedShape = decoratedShape;
    }

    // Method 2 - draw()
    // Outside abstract class
    public void draw() { decoratedShape.draw(); }
}
```

▼ Step 4. Create a concrete decorator class extending the above abstract decorator class.

Creating the RedShapeDecorator Class

```
// Class 3
// Concrete class extending the abstract class
// RedShapeDecorator.java
public class RedShapeDecorator extends ShapeDecorat
{
    public RedShapeDecorator(Shape decoratedShape)
    {
        super(decoratedShape);
    }

    @Override public void draw()
    {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }
```

```
    private void setRedBorder(Shape decoratedShape)
    {
      // Display message whenever function is calle
        System.out.println("Border Color: Red");
    }
}
```

▼ Step 5. Now use the concrete decorator class created above to decorate interface objects.

Creating the DecoratorPatternDemo Class

```
// Class
// Main class
public class DecoratorPatternDemo
{
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of Shape interface
        // inside the main() method
        Shape circle = new Circle();

        Shape redCircle
            = new RedShapeDecorator(new Circle());

        Shape redRectangle
            = new RedShapeDecorator(new Rectangle()

        // Display message
        System.out.println("Circle with normal borc

        // Calling the draw method over the
        // object calls as created in
        // above classes
```

```
        // Call 1
        circle.draw();

        // Display message
        System.out.println("\nCircle of red border'

        // Call 2
        redCircle.draw();

        // Display message
        System.out.println("\nRectangle of red borc

        // Call 3
        redRectangle.draw();
    }
}
```

Output:

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

▼ Facade Pattern

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

▼ Implementation

▼ Step 1. Create the Interface

Create the Shape Interface

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

▼ Step 2. Create the Concrete Classes implementing the Interface

Create the Rectangle Concrete Class

```
public class Rectangle implements Shape
{
   @Override
   public void draw()
     {
      System.out.println("Rectangle::draw()");
   }
}
```

Create the Square Concrete Class

```
public class Square implements Shape
{
   @Override
   public void draw()
     {
      System.out.println("Square::draw()");
```

```
   }
}
```

Create the Circle Concrete Class

```
public class Circle implements Shape
{
   @Override
   public void draw()
     {
      System.out.println("Circle::draw()");
   }
}
```

▼ Step 3. Create the Facade Class

Creating the ShapeMaker Class

```
public class ShapeMaker
{
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker()
     {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle()
     {
      circle.draw();
   }
```

```
   public void drawRectangle()
     {
      rectangle.draw();
   }

   public void drawSquare()
     {
      square.draw();
   }
}
```

▼ Step 4. Create the Runner Class to use the Facade Class

Creating the FacadePatternDemo Class

```
public class FacadePatternDemo
{
   public static void main(String[] args)
     {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

Output:

```
💡 Circle::draw()
   Rectangle::draw()
   Square::draw()
```

▼ When should this pattern be used.

The facade pattern is appropriate when you have a complex system that you want to expose to clients in a simplified way, or you want to

make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

▼ Flyweight Pattern

Flyweight pattern is one of the structural design patterns as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 10^5).

In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.

▼ Implementation

▼ Step 1. Create the Common Interface

Creating the Employee Interface

```
public interface Employee
{
    public void assignSkills(String skill);
    public void task();
}
```

▼ Step 2. Create the Concrete Methods with Task as Intrinsic Property and Skills as Extrinsic Property.

Creating the Developer Concrete Class

```
public class Developer implements Employee
{
    private final String JOB;
    private String skill;
```

```
    public Developer()
    {
        this.JOB="Test the issue";
    }

    @Override
    public void assignSkills(String skill)
    {
        this.skill=skill;
    }

    @Override
    public void task()
    {
        System.out.println("Developer with skill: '
    }
}
```

Creating the Tester Concrete Class

```
public class Tester implements Employee
{
    private final String JOB;
    private String skill;

    public Tester()
    {
        this.JOB="Fix the issue";
    }

    @Override
    public void assignSkills(String skill)
    {
        this.skill=skill;
    }
```

```
    @Override
    public void task()
    {
        System.out.println("Tester with skill: "+th
    }
}
```

▼ Step 3. Create the Object Factory.

Creating the EmployeeFactory Class

```
import java.util.HashMap;

public class EmployeeFactory
{
    private static HashMap<String, Employee> employ

    public static Employee getEmployee(String type)
    {
        Employee employee=null;
        if(employeeMap.containsKey(type))
        {
            employee=employeeMap.get(type);
        }
        else
        {
            switch (type)
            {
                case "Developer":
                {
                    System.out.println("Developer C
                    employee=new Developer();
                    break;
                }
                case "Tester":
```

```
                {
                    System.out.println("Tester Obje
                    employee=new Tester();
                    break;
                }
                default:
                    System.out.println("No such Emp
            }
            employeeMap.put(type, employee);
        }
        return employee;
    }
}
```

▼ Step 4. Create the Client to Call the Object Factory.

Creating the Engineering Class

```
import java.util.Random;

public class Engineering
{
    private static String employeeType[]= {"Develop
    private static String skills[]= {"Java","C++","

    public static void main(String[] args)
    {
        for(int i=0;i<10;i++)
        {
            Employee employee=EmployeeFactory.getEm
            employee.assignSkills(getRandSkill());
            employee.task();
        }
    }

    public static String getRandSkill()
```

```
    {
        Random random=new Random();
        int randInt=random.nextInt(skills.length);
        return skills[randInt];
    }

    public static String getRandEmployee()
    {
        Random random=new Random();
        int randInt=random.nextInt(employeeType.ler
        return employeeType[randInt];
    }
}
```

Output:

```
Tester Object Created
Tester with skill: Java with job: Fix the issue
Tester with skill: Java with job: Fix the issue
Developer Object Created
Developer with skill: Python with job: Test the issue
Tester with skill: Python with job: Fix the issue
Tester with skill: Java with job: Fix the issue
Tester with skill: Python with job: Fix the issue
Tester with skill: C++ with job: Fix the issue
Developer with skill: .Net with job: Test the issue
Tester with skill: C++ with job: Fix the issue
Tester with skill: Java with job: Fix the issue
```

▼ Why do we care for number of objects in our program?

1. Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.

2. Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

▼ Proxy Pattern

In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.
In proxy pattern, we create object having original object to interface it’s functionality to outer world.

▼ Implementation

▼ Step 1. Create the Interface that is used to implement in the Global and Filter Concrete Classes.

Creating the Internet Interface

```
public interface Internet
{
    public void connectTo(String serverhost) throws
}
```

▼ Step 2. Create the Global and Filter Classes.

Creating the Global Concrete RealInternet Class

```
public class RealInternet implements Internet
{
    @Override
    public void connectTo(String serverhost)
    {
        System.out.println("Connecting to "+ server
    }
}
```

Creating the Filter ProxyInternet Class

```
import java.util.ArrayList;
import java.util.List;
```

```
public class ProxyInternet implements Internet
{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;

    static
    {
        bannedSites = new ArrayList<String>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
        bannedSites.add("lnm.com");
    }

    @Override
    public void connectTo(String serverhost) throws
    {
        if(bannedSites.contains(serverhost.toLowerC
        {
            throw new Exception("Access Denied");
        }
        internet.connectTo(serverhost);
    }
}
```

▼ Step 3. Create the Client Class.

Creating the Client Concrete Class

```
public class Client
{
    public static void main (String[] args)
    {
        Internet internet = new ProxyInternet();
        try
        {
```

```
            internet.connectTo("geeksforgeeks.org");
            internet.connectTo("abc.com");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
Connecting to geeksforgeeks.org
Access Denied
```

▼ Advantages

1. One of the advantages of Proxy pattern is security.
2. This pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.
3. The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.

▼ Disadvantages

This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.

▼ Behavioural Design Pattern

▼ Chain Of Responsibility Pattern

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide

themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

▼ Where and When Chain of Responsibility pattern is applicable :

1. When you want to decouple a request's sender and receiver.
2. Multiple objects, determined at runtime, are candidates to handle a request.
3. When you don't want to specify handlers explicitly in your code.
4. When you want to issue a request to one of several objects without specifying the receiver explicitly.

▼ Implementation

▼ Step 1. Create the Handler abstraction, which will be used my the Concrete Handlers.

Creating the AbstractLogger abstract class

```
public abstract class AbstractLogger
{
    public static int INFO=1;
    public static int DEBUG=2;
    public static int ERROR=3;

    protected int level;

    protected AbstractLogger nextLogger;

    public void setNextLogger(AbstractLogger nextLo
    {
        this.nextLogger=nextLogger;
    }

    public void logMessage(int level,String message
    {
```

```
        if(this.level==level)
        {
            write(message);
        }
        else
        {
            if(nextLogger!=null)
            {
                nextLogger.logMessage(level, messag
            }
            else
            {
                System.out.println("Chain ended");
            }
        }
    }

    abstract protected void write(String message);
}
```

▼ Step 2. Create the Concrete Handlers.

Creating ConsoleLogger Concrete Handler

```
public class ConsoleLogger extends AbstractLogger
{
    public ConsoleLogger(int level)
    {
        this.level=level;
    }
    @Override
    protected void write(String message)
    {
        System.out.println("ConsoleLogger::"+messag
```

```
    }
}
```

Creating FileLogger Concrete Handler

```
public class FileLogger extends AbstractLogger
{
    public FileLogger(int level)
    {
        this.level=level;
    }
    @Override
    protected void write(String message)
    {
        System.out.println("FileLogger::"+message);
    }
}
```

Creating ErrorLogger Concrete Handler

```
public class ErrorLogger extends AbstractLogger
{
    public ErrorLogger(int level)
    {
        this.level=level;
    }
    @Override
    protected void write(String message)
    {
        System.out.println("ErrorLogger::"+message);
    }
}
```

▼ Step 3. Create the Client, which will access the handler.

Creating the Client Class, where the chain is established.

```
public class ChainPatternDemo
{
    private static AbstractLogger getChainLogger()
    {
        AbstractLogger errorLogger=new ErrorLogger(
        AbstractLogger fileLogger=new FileLogger(Ab
        AbstractLogger consoleLogger=new ConsoleLog

        errorLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(consoleLogger);

        return errorLogger;
    }
    public static void main(String[] args)
    {
        AbstractLogger loggerChain=getChainLogger()

        loggerChain.logMessage(AbstractLogger.INFO,
        System.out.println();

        loggerChain.logMessage(AbstractLogger.DEBUG
        System.out.println();

        loggerChain.logMessage(AbstractLogger.ERROR
        System.out.println();

        loggerChain.logMessage(5, "Out of bounds me
    }
}
```

Output:

ConsoleLogger::This is information message

FileLogger::This is debug message

ErrorLogger::This is error message

Chain ended

▼ Advantages

1. To reduce the coupling degree. Decoupling it will request the sender and receiver.
2. Simplified object. The object does not need to know the chain structure.
3. Enhance flexibility of object assigned duties. By changing the members within the chain or change their order, allow dynamic adding or deleting responsibility.
4. Increase the request processing new class of very convenient

▼ Disadvantages

1. The request must be received not guarantee.
2. The performance of the system will be affected, but also in the code debugging is not easy may cause cycle call.
3. It may not be easy to observe the characteristics of operation, due to debug.

▼ Command Pattern

Command pattern is a data driven design pattern and falls under behavioural pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

The command pattern encapsulates a request as an object, thereby letting us parameterise other objects with different requests, queue or

log requests, and support undoable operations.

▼ Implementation

▼ Step 1. Create a Command Interface.

Creating the Order Interface

```
public interface Order
{
    public void execute();
}
```

▼ Step 2. Create a Request Class.

Creating the Stock Class

```
public class Stock
{
    private String name="ABC";
    private int quantity=10;

    public void buy()
    {
        System.out.println("Stock [Name: "+name+",
    }

    public void sell()
    {
        System.out.println("Stock [Name: "+name+",
    }
}
```

▼ Step 3. Creating Concrete Classes after implementing the Command Interface.

Creating the BuyStock Concrete Class

```
public class BuyStock implements Order
{
    private Stock stock;

    public BuyStock(Stock stock)
    {
        this.stock=stock;
    }
    @Override
    public void execute()
    {
        stock.buy();
    }
}
```

Creating the SellStock Concrete Class

```
public class SellStock implements Order
{
    private Stock stock;

    public SellStock(Stock stock)
    {
        this.stock=stock;
    }
    @Override
    public void execute()
    {
        stock.sell();
    }
}
```

▼ Step 4. Create a Command Invoker class.

Creating the Broker Class

```
import java.util.ArrayList;
import java.util.List;

public class Broker
{
    private List<Order> orderList=new ArrayList<Orc

    public void takeOrder(Order order)
    {
        orderList.add(order);
    }

    public void placeOrders()
    {
        for(Order order:orderList)
        {
            order.execute();
        }
        orderList.clear();
    }
}
```

▼ Step 5. Create a Client Class to take and execute commands.

Creating the CommandPatternDemo Client Class

```
public class CommandPatternDemo
{
    public static void main(String[] args)
    {
        Stock stock=new Stock();

        BuyStock buyStock=new BuyStock(stock);
        SellStock sellStock=new SellStock(stock);
```

```
        Broker broker=new Broker();
        broker.takeOrder(buyStock);
        broker.takeOrder(sellStock);

        broker.placeOrders();
    }
}
```

Output:

```
Stock [Name: ABC, Quantity: 10 ] bought
Stock [Name: ABC, Quantity: 10 ] sold
```

▼ Advantages

1. Makes our code extensible as we can add new commands without changing existing code.
2. Reduces coupling between the invoker and receiver of a command.

▼ Disadvantages

1. Increase in the number of classes for each individual command.

▼ Interpreter Pattern

Interpreter design pattern is one of the behavioural design pattern. Interpreter pattern is used to defines 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. This pattern is used in SQL parsing, symbol processing engine etc.
- This pattern performs upon a hierarchy of expressions. Each expression here is a terminal or non-terminal.
- The tree structure of Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal

expressions being leaf objects and non-terminal expressions being composites.

- The tree contains the expressions to be evaluated and is usually generated by a parser. The parser itself is not a part of the interpreter pattern.

▼ Implementation

▼ Step 1. Create the Abstract Expression

Creating the Expression Interface

```
public interface Expression
{
    boolean interpreter(String con);
}
```

▼ Step 2. Create the Terminal Expression

Creating the TerminalExpression Class

```
public class TerminalExpression implements Expressi
{
    String data;

    public TerminalExpression(String data)
    {
        this.data=data;
    }

    @Override
    public boolean interpreter(String con)
    {
        boolean returnData=false;
        if(con.contains(data))
        {
            returnData=true;
        }
```

```
        else
        {
            returnData=false;
        }
        return returnData;
    }
}
```

▼ Step 3. Create the Non-Terminal Expressions

Creating the OrExpression Class

```
public class OrExpression implements Expression
{
    Expression expression1;
    Expression expression2;

    public OrExpression(Expression expression1,Expr
    {
        this.expression1=expression1;
        this.expression2=expression2;
    }

    @Override
    public boolean interpreter(String con)
    {
//      boolean eval1=expression1.interpreter(con);
//      boolean eval2=expression2.interpreter(con);
        return expression1.interpreter(con)||expres
    }
}
```

Creating the AndExpression Class

```
public class AndExpression implements Expression
{
```

```
    Expression expression1;
    Expression expression2;

    public AndExpression(Expression expression1,Exp
    {
        this.expression1=expression1;
        this.expression2=expression2;
    }

    @Override
    public boolean interpreter(String con)
    {
//      boolean eval1=expression1.interpreter(con);
//      boolean eval2=expression2.interpreter(con);
        return expression1.interpreter(con)&&expres
    }
}
```

▼ Step 4. Create the Client, which is basically the Expression Parser

Creating the InterpreterDesignPattern Class, which will be basically the runner class.

```
public class InterpreterDesignPattern
{
    public static void main(String[] args)
    {
        Expression person1=new TerminalExpression('
        Expression person2=new TerminalExpression('
        Expression isSingle=new OrExpression(perso

        Expression vikram=new TerminalExpression("V
        Expression committed=new TerminalExpressio
        Expression isCommitted=new AndExpression(vi

        System.out.println(isSingle.interpreter("Ku
```

```
        System.out.println(isSingle.interpreter("Lo
        System.out.println(isSingle.interpreter("Ad

        System.out.println(isCommitted.interpreter(
        System.out.println(isCommitted.interpreter(
    }
}
```

Output:

```
true
true
false
true
false
```

▼ Advantages

- It’s easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.
- Implementing the grammar is easy, too. Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and often their generation can be automated with a compiler or parser generator.

▼ Disadvantages

- Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar. Hence grammars containing many rules can be hard to manage and maintain.

▼ Iterator Pattern

The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.

Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.

▼ Implementation

▼ Step 1. Create the Collection and Iterator Abstraction.

Create the Collection Interface

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

Create the Iterator Interface

```
public interface Iterator
{
    boolean hasNext();

    Object next();
}
```

▼ Step 2. Create the Concrete Class for the Object for the Collection.

Create the Notification Class

```
public class Notification
{
    String notification;

    public Notification(String notification)
```

```
    {
        this.notification=notification;
    }

    public String getNotification()
    {
        return notification;
    }
}
```

▼ Step 3. Create the Concrete Classes for the Abstractions already defined.

Creating the NotificationCollection Class

```
public class NotificationCollection implements Coll
{
    static final int MAX_ITEMS=6;
    int numberOfItems=0;
    Notification[] notificationList;

    public NotificationCollection()
    {
        notificationList=new Notification[MAX_ITEMS
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }

    public void addItem(String str)
    {
        Notification notification=new Notification(
        if(numberOfItems>=MAX_ITEMS)
        {
            System.out.println("Collection is Full"
        }
```

```
        else
        {
            notificationList[numberOfItems]=notific
            numberOfItems+=1;
        }
    }

    @Override
    public Iterator createIterator()
    {
        return new NotificationIterator(notificatio
    }
}
```

Creating the NotificationIterator Class

```
public class NotificationIterator implements Iterat
{
    Notification[] notificationList;
    int pos=0;

    public NotificationIterator(Notification[] noti
    {
        this.notificationList=notificationList;
    }

    @Override
    public Object next()
    {
        Notification notification=notificationList[
        pos+=1;
        return notification;
    }

    @Override
    public boolean hasNext()
```

```
    {
        boolean returnVal=false;
        if(pos>=notificationList.length||notificati
        {
            returnVal=false;
        }
        else
        {
            returnVal=true;
        }
        return returnVal;
    }
}
```

▼ Step 4. Create the Client and Implementation

Creating the NotificationBar Class

```
public class NotificationBar
{
    NotificationCollection notifications;

    public NotificationBar(NotificationCollection n
    {
        this.notifications=notifications;
    }

    public void printNotification()
    {
        Iterator iterator=notifications.createItera
        System.out.println("--Notifications--");
        while (iterator.hasNext())
        {
            Notification notification=(Notificatior
            System.out.println(notification.getNoti
        }
```

```
    }
}
```

Creating the IteratorDesignPattern Class

```
public class IteratorDesignPattern
{
    public static void main(String[] args)
    {
        NotificationCollection notificationCollecti
        NotificationBar nb=new NotificationBar(noti
        nb.printNotification();
    }
}
```

Output:

```
—Notifications—
Notification 1
Notification 2
Notification 3
```

▼ Advantages

▼ Easy to Implement and Readable

- The iterator design pattern is easy to implement it uses iterator interface, container interface and concrete class that implement container interface.
- Another class that implements iterator interface (provides implementation ways for collection traversal) and finally client class.

▼ Single Responsibility Principle

This design pattern follows Single Responsibility Principle; the Single Responsibility Principle allows us to clean up the client and

collections of the traversal algorithms into separate classes.

▼ The Open/Closed Principle

The Open/Closed Principle allows implementation of new types of collections and iterators without breaking anything.

▼ Clean Code

Clean code because the client/context does not use a complex interface and the system is more flexible and reusable.

▼ Parallel Iteration

You can iterate over the same collection in parallel because each iterator object contains its own iteration state.

▼ Disadvantages

▼ Performance

This pattern is less efficient going through elements of some specialised collections directly, Uses more memory than direct element access.

▼ Not Suggested for Simple Collections

Using this design patterns with simple collections can be an overkill if your application.

▼ Mediator Pattern

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Mediator pattern falls under behavioural pattern category.

▼ Implementation

▼ Step 1. Implement the Concrete Classes where there is no hard coupling of classes.

Creating the ChatRoom Class

```
import java.util.Date;

public class ChatRoom
{
    public static void showMessage(User user,String
    {
        System.out.println(new Date().toString()+"
    }
}
```

Creating the User Class

```
public class User
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public User(String name)
    {
        this.name=name;
    }

    public void sendMessage(String message)
    {
        ChatRoom.showMessage(this, message);
```

```
    }
}
```

▼ Step 2. Call the Mediator Class from the Client.

Creating the MediatorPatterDemo Concrete Class

```
public class MediatorPatternDemo
{
    public static void main(String[] args)
    {
        User user1=new User("Adipta");
        User user2=new User("Aritra");

        user1.sendMessage("Hello I am user1");
        user2.sendMessage("Hello I am user2");
    }
}
```

Output

```
💡 Mon Jul 18 19:22:43 IST 2022 [Adipta]: Hello I am user1
   Mon Jul 18 19:22:43 IST 2022 [Aritra]: Hello I am user2
```

▼ Advantage

It limits subclassing. A mediator localises behaviour that otherwise would be distributed among several objects. Changing this behaviour requires subclassing Mediator only, Colleague classes can be reused as is.

▼ Disadvantage

It centralises control. The mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that’s hard to maintain

▼ Memento Pattern

Memento pattern is a behavioural design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

▼ Implementation

▼ Step 1. Create the Memento, i.e. the object that is going to maintain the state of the originator.

Creating the Memento Class

```
public class Memento
{
    private String state;

    public Memento(String state)
    {
        this.state=state;
    }

    public String getState()
    {
        return this.state;
    }
}
```

▼ Step 2. Create the Originator, i.e. the object for which the state is to be saved.

Creating the Originator Class

```
public class Originator
{
    private String state;

    public void setState(String state)
```

```
    {
        this.state=state;
    }

    public String getState()
    {
        return this.state;
    }

    public Memento saveStateToMemento()
    {
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento
    {
        this.state=memento.getState();
    }
}
```

▼ Step 3. Create the CareTaker, whose object keeps the track of multiple Memento Objects, which is like save-points.

Creating the CareTaker Class

```
import java.util.ArrayList;
import java.util.List;

public class CareTaker
{
    private List<Memento> mementoList=new ArrayList

    public void add(Memento state)
    {
        mementoList.add(state);
    }
```

```
    public Memento get(int index)
    {
        return mementoList.get(index);
    }
}
```

▼ Step 4. Create a Client to Check the Functionality.

Creating the MementoPatternDemo Class

```
public class MementoPatternDemo
{
    public static void main(String[] args)
    {
        Originator originator=new Originator();
        CareTaker careTaker=new CareTaker();

        originator.setState("State 1");
        originator.setState("State 2");
        careTaker.add(originator.saveStateToMemento

        originator.setState("State 3");
        careTaker.add(originator.saveStateToMemento

        originator.setState("State 4");
        System.out.println("Current State:"+origina

        originator.getStateFromMemento(careTaker.ge
        System.out.println("First Saved State:"+ori
        originator.getStateFromMemento(careTaker.ge
        System.out.println("Second Saved State:"+or
    }
}
```

Output:

```
💡 Current State:State 4
First Saved State:State 2
Second Saved State:State 3
```

▼ Advantage

We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.

▼ Disadvantage

If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

▼ Observer Pattern

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

▼ Implementation

▼ Step 1. Create the Interfaces for Observer and Subject.

Creating the Subject Interface

```
public interface Observer
{

    void update();

    void subscribeChannel(Channel channel);

}
```

Creating the Observer Interface

```
public interface Subject
{
```

```
    void subscribe(Subscriber subscriber);

    void unSubscribe(Observer subscriber);

    void notifySubscribers();

    void upload(String title);

}
```

▼ Step 2. Create the Concrete Classes implementing the Interfaces.

Creating the Channel Class

```
import java.util.ArrayList;
import java.util.List;

public class Channel implements Subject
{
    List<Subscriber> subscribers=new ArrayList<Subs
    public String title;

    @Override
    public void subscribe(Subscriber subscriber)
    {
        subscribers.add(subscriber);
    }

    @Override
    public void unSubscribe(Observer subscriber)
    {
        subscribers.remove(subscriber);
    }

    @Override
```

```
    public void notifySubscribers()
    {
        for(Observer subscriber:subscribers)
        {
            subscriber.update();
        }
    }

    @Override
    public void upload(String title)
    {
        this.title=title;
        notifySubscribers();
    }
}
```

Creating the Subscriber Class

```
public class Subscriber implements Observer
{
    private String nameOfSubscriber;
    private Channel channel=new Channel();

    public Subscriber(String nameOfSubscriber)
    {
        this.nameOfSubscriber=nameOfSubscriber;
    }

    @Override
    public void update()
    {
        System.out.println("Hello "+this.nameOfSubs
    }

    @Override
    public void subscribeChannel(Channel channel)
```

```
    {
        this.channel=channel;
    }
}
```

▼ Step 3. Create the Runner Class.

Creating the Youtube Class

```
public class Youtube
{
    public static void main(String[] args)
    {
        Channel channel=new Channel();

        Subscriber s1=new Subscriber("Adipta");
        Subscriber s2=new Subscriber("Aritra");
        Subscriber s3=new Subscriber("Sagnik");
        Subscriber s4=new Subscriber("Akshay");
        Subscriber s5=new Subscriber("Arka");

        channel.subscribe(s1);
        channel.subscribe(s2);
        channel.subscribe(s3);
        channel.subscribe(s4);
        channel.subscribe(s5);

        channel.unSubscribe(s3);

        s1.subscribeChannel(channel);
        s2.subscribeChannel(channel);
        s3.subscribeChannel(channel);
        s4.subscribeChannel(channel);
        s5.subscribeChannel(channel);

        channel.upload("Learning Observer Design Pa
```

```
    }
}
```

Output:

```
Hello Adipta, video uploaded: Learning Observer Design Pattern
Hello Aritra, video uploaded: Learning Observer Design Pattern
Hello Akshay, video uploaded: Learning Observer Design Pattern
Hello Arka, video uploaded: Learning Observer Design Pattern
```

▼ Advantages

Provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

Observer pattern provides this loose coupling as:

- Subject only knows that observer implement Observer interface.Nothing more.
- There is no need to modify Subject to add or remove observers.
- We can reuse subject and observer classes independently of each other.

▼ Disadvantages

- Memory leaks caused by Lapsed listener problem because of explicit register and unregistering of observers.

▼ When to Use this Pattern

You should consider using this pattern in your application when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.

▼ Real Life Uses

- It is heavily used in GUI toolkits and event listener. In java the button(subject) and onClickListener(observer) are modelled with observer pattern.
- Social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
- All users of an app on play store gets notified if there is an update.

▼ State Pattern

State pattern is one of the behavioural design pattern. State design pattern is used when an Object changes its behavioural based on its internal state.

If we have to change behaviour of an object based on its state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.

▼ Implementation

▼ Step 1. Create the State, that defines the interface for declaring what each concrete state should do.

Creating the MobileAlertState Interface

```
public interface MobileAlertState
{
    public void alert(AlertStateContext alertStateC
}
```

▼ Step 2. Create the Concrete Classes, that provide the implementation for the methods defined in the state.

Creating the Silent Class

```
public class Silent implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext alertState(
    {
        System.out.println("Silent...");
    }
}
```

Creating the Vibration Class

```
public class Vibration implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext alertState(
    {
        System.out.println("Vibration...");
    }
}
```

▼ Step 3. Create the Context, which defines the class to interact with the client. It maintains the reference to the concrete state object.

Creating the AlertStateContext Class

```
public class AlertStateContext
{
    private MobileAlertState currentState;

    public AlertStateContext()
    {
        this.currentState=new Vibration();
    }
```

```
    public void setState(MobileAlertState currentSt
    {
        this.currentState=currentState;
    }

    public void alert()
    {
        currentState.alert(this);
    }
}
```

▼ Step 4. Call the Context to check the Context like the Client.

Creating the StatePatternDemo Class

```
public class StatePatternDemo
{
    public static void main(String[] args)
    {
        AlertStateContext stateContext=new AlertSta

        stateContext.alert();
        stateContext.alert();

        stateContext.setState(new Silent());

        stateContext.alert();
        stateContext.alert();
        stateContext.alert();
    }
}
```

Output:

Vibration...
Vibration...
Silent...
Silent...
Silent...

▼ Advantages

- With State pattern, the benefits of implementing polymorphic behavior are evident, and it is also easier to add states to support additional behavior.
- In the State design pattern, an object's behavior is the result of the function of its state, and the behavior gets changed at runtime depending on the state. This removes the dependency on the if/else or switch/case conditional logic. For example, in the TV remote scenario, we could have also implemented the behavior by simply writing one class and method that will ask for a parameter and perform an action (switch the TV on/off) with an if/else block.
- The State design pattern also improves Cohesion since state-specific behaviors are aggregated into the ConcreteState classes, which are placed in one location in the code.

▼ Disadvantages

- The State design pattern can be used when we need to change state of object at runtime by inputting in it different subclasses of some State base class. This circumstance is advantage and disadvantage in the same time, because we have a clear separate State classes with some logic and from the other hand the number of classes grows up.

▼ Strategy Pattern

In computer programming, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behaviour to be selected at runtime.

▼ Implementation

▼ Step 1. Create the Interface for the Strategy.

Creating the Strategy Interface

```
public interface Strategy
{
    public int doOperation(int num1,int num2);
}
```

▼ Step 2. Create the Concrete Classes implementing the Strategy Interface.

Creating the OperationAdd Class

```
public class OperationAdd implements Strategy
{
    @Override
    public int doOperation(int num1, int num2)
    {
        return num1+num2;
    }
}
```

Creating the OperationMultiply Class

```
public class OperationMultiply implements Strategy
{
    @Override
    public int doOperation(int num1, int num2)
    {
        return num1*num2;
    }
}
```

Creating the OperationSubstract Class

```
public class OperationSubstract implements Strategy
{
    @Override
    public int doOperation(int num1, int num2)
    {
        return num1-num2;
    }
}
```

▼ Step 3. Create the Context Class, which the Client Class will call.

Creating the Context Class

```
public class Context
{
    private Strategy strategy;
    public Context(Strategy strategy)
    {
        this.strategy=strategy;
    }

    public int executeStrategy(int num1,int num2)
    {
        return strategy.doOperation(num1, num2);
    }
}
```

▼ Step 4. Create the Client Class, that will call the Client Class.

Creating the StrategyPatternDemo Class

```
public class StrategyPatternDemo
{
    public static void main(String[] args)
    {
        Context context = new Context(new Operation
```

```
        System.out.println("10 + 5 = "+context.exec

        context = new Context(new OperationSubstrac
        System.out.println("10 - 5 = "+context.exec

        context = new Context(new OperationMultiply
        System.out.println("10 * 5 = "+context.exec
    }
}
```

Output:

```
💡 10 + 5 = 15
   10 - 5 = 5
   10 * 5 = 50
```

▼ Advantages

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

▼ Disadvantages

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

▼ Template Pattern

Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm are preserved by the parent class.

Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps. These steps can be an abstract method that will be implemented by its subclasses.

▼ Implementation

▼ Step 1. Create the Abstract Class that contains the template method. This method should be made final, so that this cannot be overridden.

Creating the OrderProcessTemplate Class

```
public abstract class OrderProcessTemplate
{
    public boolean isGift;

    public abstract void doSelect();

    public abstract void doPayment();
```

```
    public final void giftWrap()
    {
        try
        {
            System.out.println("Gift Wrap Successfu
        }
        catch(Exception ex)
        {
            System.out.println("Gift Wrap Unsuccess
        }
    }

    public abstract void doDelivery();

    public final void processOrder(boolean isGift)
    {
        doSelect();
        doPayment();
        if(isGift)
        {
            giftWrap();
        }
        doDelivery();
    }
}
```

▼ Step 2. Create the Concrete Class that implements the operations required by the template method that were defined as abstract in the parent class.

Creating the NetOrder Class

```
public class NetOrder extends OrderProcessTemplate
{
    @Override
```

```
    public void doSelect()
    {
        System.out.println("Item added to online sh
        System.out.println("Get gift wrap preferenc
        System.out.println("Get delivery address.")
    }

    @Override
    public void doPayment()
    {
        System.out.println("Online payment through
    }

    @Override
    public void doDelivery()
    {
        System.out.println("Ship the item throught
    }
}
```

Creating the StoreOrder Class

```
public class StoreOrder extends OrderProcessTemplat
{
    @Override
    public void doSelect()
    {
        System.out.println("Customer chooses the it
    }

    @Override
    public void doPayment()
    {
        System.out.println("Pays at counter through
    }
```

```
    @Override
    public void doDelivery()
    {
        System.out.println("Items delivered to deli
    }
}
```

▼ Step 3. Create the Client to test the implementation.

Creating the TemplateDesignPatternDemo Class

```
public class TemplateDesignPatternDemo
{
    public static void main(String[] args)
    {
        OrderProcessTemplate netOrder=new NetOrder(
        netOrder.processOrder(true);

        System.out.println();

        OrderProcessTemplate storeOrder=new StoreOr
        storeOrder.processOrder(true);
    }
}
```

Output:

💡 Item added to online shopping cart.
Get gift wrap preference.
Get delivery address.
Online payment through Netbanking, card or Paytm.
Gift Wrap Successful
Ship the item throught post to delivery address.

Customer chooses the item from shelf.
Pays at counter through cash/POS.
Gift Wrap Successful
Items delivered to delivery counter.

▼ When to use Template Method

The template method is used in frameworks, where each implements the invariant parts of a domain's architecture, leaving "placeholders" for customisation options.

The template method is used for the following reasons :

- Let subclasses implement varying behavior (through method overriding)
- Avoid duplication in the code, the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses.
- Control at what points subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

▼ Visitor Pattern

Visitor design pattern is one of the behavioural design patterns. It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

▼ Implementation

▼ Step 1. Create the interface for the visitable class, that will implement the Accept() method, and an interface for the visitor class, that will implement the Visit() method.

Creating the ComputerPart Concrete Interface

```
public interface ComputerPart
{
    public void accept(ComputerPartVisitor computer
}
```

Creating the ComputerPartVisitor Interface

```
public interface ComputerPartVisitor
{
    public void visit(Keyboard keyboard);
    public void visit(Monitor monitor);
    public void visit(Mouse mouse);
    public void visit(Computer computer);
}
```

▼ Step 2. Create the Visitor Concrete Classes

Creating the Mouse Concrete Class

```
public class Mouse implements ComputerPart
{
    @Override
    public void accept(ComputerPartVisitor computer
    {
        computerPartVisitor.visit(this);
    }
}
```

Creating the Keyboard Concrete Class

```
public class Keyboard implements ComputerPart
{
    @Override
    public void accept(ComputerPartVisitor computer
    {
        computerPartVisitor.visit(this);
    }
}
```

Creating the Monitor Concrete Class

```
public class Monitor implements ComputerPart
{
    @Override
    public void accept(ComputerPartVisitor computer
    {
        computerPartVisitor.visit(this);
    }
}
```

Creating the Computer Concrete Class

```
public class Computer implements ComputerPart
{
    ComputerPart[] computerParts;

    public Computer()
    {
        computerParts=new ComputerPart[] {new Mouse
    }

    @Override
    public void accept(ComputerPartVisitor computer
    {
        for(int i=0;i<computerParts.length;i++)
```

```
        {
            computerParts[i].accept(computerPartVis
        }
        computerPartVisitor.visit(this);
    }
}
```

▼ Step 3. Create the Visitable Concrete Classes

Creating the ComputerPartDisplayVisitor Class

```
public class ComputerPartDisplayVisitor implements
{
    @Override
    public void visit(Keyboard keyboard)
    {
        System.out.println("Displaying Keyboard");
    }

    @Override
    public void visit(Monitor monitor)
    {
        System.out.println("Displaying Monitor");
    }

    @Override
    public void visit(Mouse mouse)
    {
        System.out.println("Displaying Mouse");
    }

    @Override
    public void visit(Computer computer)
    {
        System.out.println("Displaying Computer");
```

```
    }
}
```

▼ Step 4. Creating the Client Class to check

Creating the VisitorPatternDemo Class

```
public class VisitorPatternDemo
{
    public static void main(String[] args)
    {
        ComputerPart computer=new Computer();
        computer.accept(new ComputerPartDisplayVisi
    }
}
```

Output:

```
Displaying Mouse
Displaying Keyboard
Displaying Monitor
Displaying Computer
```

▼ Advantages

- If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

▼ Disadvantages

- We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.

- If there are too many implementations of visitor interface, it makes it hard to extend.

JEE or J2EE Design Patterns

Presentation Layer Design Pattern

Intercepting Filter Pattern

Preprocessing and post-processing of a request refer to actions taken before and after the core processing of that request. Some of these actions determine whether processing will continue, while others manipulate the incoming or outgoing data stream into a form suitable for further processing.

The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

Implementation

Step 1. Create an abstraction for the Filters.

Creating the Filter Interface.

```
public interface Filter
{
    public void execute(String request);
}
```

Step 2. Create the Concrete Classes for the Filter Abstraction.

Creating the AuthenticationFilter Concrete Class.

```
public class AuthenticationFilter implements Filter
{
    @Override
    public void execute(String request)
    {
        System.out.println("In AuthenticationFilter
    }
}
```

Creating the DebugFilter Concrete Class.

```
public class DebugFilter implements Filter
{
    @Override
    public void execute(String request)
    {
        System.out.println("In DebugFilter: "+reque
    }
}
```

▼ Step 3. Create the Target Concrete Class, that is the resource requested by the Client.

Creating the Target Concrete Class.

```
public class Target
{
    public void execute(String request)
    {
        System.out.println("In target: "+request);
    }
}
```

▼ Step 4. Create the FilterChain Concrete Class which is an ordered collection of independent filters.

Creating the FilterChain Concrete Class.

```
import java.util.ArrayList;
import java.util.List;

public class FilterChain
{
    private List<Filter> filters=new ArrayList<Filt
    private Target target;

    public void addFilter(Filter filter)
    {
        filters.add(filter);
    }

    public void setTarget(Target target)
    {
        this.target=target;
    }

    public void execute(String request)
    {
        for(Filter filter:filters)
        {
            filter.execute(request);
        }
        target.execute(request);
    }
}
```

▼ Step 5. Create the FilterManager Concrete Class, that manages the filter processing, creates the FilterChain with the appropriate filters in the correct order.

Creating the FilterManager Concrete Class.

```
public class FilterManager
{
    private FilterChain filterChain;

    public FilterManager(Target target)
    {
        filterChain=new FilterChain();
        filterChain.setTarget(target);
    }

    public void setFilter(Filter filter)
    {
        filterChain.addFilter(filter);
    }

    public void filterRequest(String request)
    {
        filterChain.execute(request);
    }
}
```

▼ Step 6. Create the Client, that wraps the FilterManager

Creating the Client Concrete Class

```
public class Client
{
    private FilterManager filterManager;

    public void setFilterManager(FilterManager filt
    {
        this.filterManager=filterManager;
    }

    public void sendRequest(String request)
```

```
    {
        filterManager.filterRequest(request);
    }
}
```

▼ Step 7. Create the Runner Class, test the implementation.

Creating InterceptingFilterDesignPattern Concrete Class.

```
public class InterceptingFilterDesignPattern
{
    public static void main(String[] args)
    {
        Target target=new Target();
        FilterManager filterManager=new FilterManag

        Filter authenticationFilter=new Authenticat
        Filter debugFilter=new DebugFilter();
        filterManager.setFilter(authenticationFilte
        filterManager.setFilter(debugFilter);

        Client client=new Client();
        client.setFilterManager(filterManager);
        client.sendRequest("Download Request");
    }
}
```

Output:

```
💡 In AuthenticationFilter: Download Request
   In DebugFilter: Download Request
   In target: Download Request
```

▼ Advantages

▼ Improved Reusability

Common code is centralised in pluggable components enhancing reuse.

▼ Increased Flexibility

Generic common components can be applied and removed declaratively, improving flexibility.

▼ Disadvantages

1. Information sharing is inefficient in intercepting pattern.

▼ Front Controller Pattern

The front controller design pattern means that all requests that come for a resource in an application will be handled by a single handler and then dispatched to the appropriate handler for that type of request. The front controller may use other helpers to achieve the dispatching mechanism.

▼ Implementation

▼ Step 1. Create the Concrete Classes for the Views, where the view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.

Creating the StudentView Class.

```
public class StudentView
{
    public void display()
    {
        System.out.println("In StudentView");
    }
}
```

Creating the TeacherView Class.

```
public class TeacherView
{
    public void display()
    {
```

```
        System.out.println("In TeacherView");
    }
}
```

▼ Step 2. Create the Dispatcher Concrete Class, where the dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource.

Creating the Dispatching Concrete Class

```
public class Dispatching
{
    private StudentView studentView;
    private TeacherView teacherView;

    public Dispatching()
    {
        this.studentView=new StudentView();
        this.teacherView=new TeacherView();
    }

    public void dispatch(String request)
    {
        if("TEACHER".equalsIgnoreCase(request))
        {
            teacherView.display();
        }
        else
        {
            studentView.display();
        }
    }
}
```

▼ Step 3. Create the Controller Concrete Class, where the Controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorisation of a user or to initiate contact retrieval.

Creating the FrontController Concrete Class.

```
public class FrontController
{
    private Dispatching dispatching;

    public FrontController()
    {
        this.dispatching=new Dispatching();
    }

    private boolean isAuthenticUser()
    {
        System.out.println("Authentication Successf
        return true;
    }

    private void trackRequest(String request)
    {
        System.out.println("Tracking request :"+req
    }

    public void dispatchRequest(String request)
    {
        trackRequest(request);
        if(isAuthenticUser())
        {
            dispatching.dispatch(request);
        }
    }
}
```

▼ Step 4. Create a Runner Class, to check the implementation.

Creating the FrontControllerPattern Class

```
public class FrontControllerPattern
{
    public static void main(String[] args)
    {
        FrontController frontController=new FrontCo
        frontController.dispatchRequest("Teacher");
        System.out.println();
        frontController.dispatchRequest("Student");
    }
}
```

Output:

```
Tracking request :Teacher view.
Authentication Successful.
In TeacherView

Tracking request :Student view.
Authentication Successful.
In StudentView
```

▼ Advantages

▼ Centralised Control

Front controller handles all the requests to the Web application. This implementation of centralised control that avoids using multiple controllers is desirable for enforcing application-wide policies such as users tracking and security.

▼ Thread-Safety

A new command object arises when receiving a new request and the command objects are not meant to be thread-safe. Thus, it

will be safe in the command classes. Though safety is not guaranteed when threading issues are gathered, codes that act with the command are still thread safe.

▼ Disadvantages

1. It is not possible to scale an application using a front controller.
2. Performance is better if you deal with a single request uniquely.

▼ View Helper Pattern

View Helper Design Pattern separates the static view such as JSPs from the processing of the business model data. The View Helper pattern is used in the presentation layer by adapting the model data and the View components. View Helper can format the model data according to the your business requirement, but it can not generate model data for the business.

In order to adapt model data into the presentation layer of the application, the view helper pattern is used, which specifies the use of helpers for such implementation. The presentation layer, generally, have a number of JavaServer (JSP) Pages. In order to present content to the user, these pages contain HTML and images. The problem takes roots when the JSP pages are required to display dynamic content which is stored within the model. The embedding of Java code needs to be avoided in order to display model data by these pages. Instead if that, helpers must be employed to resolve the issue. There are certain choices that the developer has, they can add JDP scriplets by embedding Java code, they can use EL or they can add helpers in order to extract the data for them. It makes more sense to employ a number of helpers in order to adapt the model to the presentation that it does if the java code is cluttered with the presentation code. This helps in keeping the presentation code and the business logic separated.

▼ Composite View Pattern

Composite View Design Pattern is a design pattern, where the main view is separated in multiple smaller views.

▼ Business Layer Design Pattern

▼ Business Delegate Pattern

The Business Delegate acts as a client-side business abstraction, it provides an abstraction for, and thus hides, the implementation of the business services. It reduces the coupling between presentation-tier clients and the system's Business services.

▼ Implementation

▼ Step 1. Create the Abstraction for the Concrete Services.

Creating the BusinessService Concrete Class

```
public interface BusinessService
{
    public void doProcess();
}
```

▼ Step 2. Create the Concrete Service Classes.

Creating the OneService Concrete Class.

```
public class OneService implements BusinessService
{
    @Override
    public void doProcess()
    {
        System.out.println("Processed from OneServi
    }
}
```

Creating the TwoService Concrete Class.

```
public class TwoService implements BusinessService
{
    @Override
    public void doProcess()
    {
        System.out.println("Processed from TwoServi
```

```
    }
}
```

▼ Step 3. Create the LookUp Service, whose object is responsible to get relative business implementation and provide business object access to business delegate object.

Creating the BusinessLookUp Concrete Class

```
public class BusinessLookUp
{
    public BusinessService getBusinessService(Strir
    {
        if("ONE".equalsIgnoreCase(serviceType))
        {
            return new OneService();
        }
        else
        {
            return new TwoService();
        }
    }
}
```

▼ Step 4. Create the Business Delegate Class, which is a single entry point class for client entities to provide access to Business Service methods.

Creating the BusinessDelegate Class.

```
public class BusinessDeligate
{
    private BusinessLookUp businessLookUp=new Busir
    private BusinessService businessService;
    private String serviceType;

    public void setServiceType(String serviceType)
```

```
    {
        this.serviceType=serviceType;
    }

    public void doTask()
    {
        businessService=businessLookUp.getBusinessS
        businessService.doProcess();
    }
}
```

▼ Step 5. Create the Client Class, which wraps the Business Delegate Object.

Creating the Client Concrete Class.

```
public class Client
{
    private BusinessDeligate businessDeligate;

    public Client(BusinessDeligate businessDeligate
    {
        this.businessDeligate=businessDeligate;
    }

    public void doTask()
    {
        businessDeligate.doTask();
    }
}
```

▼ Step 6. Create the Runner Class, to test the implementation.

Creating the BusinessDeligatePattern Class

```
public class BusinessDeligatePattern
{
```

```
    public static void main(String[] args)
    {
        BusinessDeligate businessDeligate=new Busi
        businessDeligate.setServiceType("One");

        Client client=new Client(businessDeligate)
        client.doTask();

        System.out.println();

        businessDeligate.setServiceType("Two");
        client.doTask();
    }
}
```

Output:

```
Processed from OneService

Processed from TwoService
```

▼ Advantages

- Business Delegate reduces coupling between presentation-tier clients and Business services.
- The Business Delegate hides the underlying implementation details of the Business service.

▼ Disadvantages

- Maintenance due the extra layer that increases the number of classes in the application.

▼ Service Locator Pattern

The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central

registry known as the “service locator” which on request returns the information necessary to perform a certain task.

The ServiceLocator is responsible for returning instances of services when they are requested for by the service consumers or the service clients.

▼ Implementations

▼ Step 1. Create the Abstraction for the Service Classes.

Creating the Service Class.

```
public interface Service
{
    public String getName();
    public void execute();
}
```

▼ Step 2. Create the BusinessService Concrete Classes. The BusinessService is a role that is fulfilled by the service the client is seeking to access. The BusinessService object is created or looked up or removed by the ServiceFactory. The BusinessService object in the context of an EJB application is an enterprise bean.

Creating the ServiceOne Class.

```
public class ServiceOne implements Service
{
    @Override
    public String getName()
    {
        return "ServiceOne";
    }

    @Override
    public void execute()
    {
        System.out.println("Executing ServiceOne");
```

```
    }
}
```

Creating the ServiceTwo Class.

```
public class ServiceTwo implements Service
{
    @Override
    public String getName()
    {
        return "ServiceTwo";
    }

    @Override
    public void execute()
    {
        System.out.println("Executing ServiceTwo");
    }
}
```

▼ Step 3. Create the Initial Context Class, the InitialContext object is the start point in the lookup and creation process. Service providers provide the context object, which varies depending on the type of business object provided by the Service Locator’s lookup and creation service.

Creating the InitialContext Class.

```
public class InitialContext
{
    public Object lookup(String name)
    {
        if("SERVICEONE".equalsIgnoreCase(name))
        {
            System.out.println("Creating a new Serv
            return new ServiceOne();
```

```
        }
        else if("SERVICETWO".equalsIgnoreCase(name)
        {
            System.out.println("Creating a new Serv
            return new ServiceTwo();
        }
        else
        {
            return null;
        }
    }
}
```

▼ Step 4. Create the Cache, which is the ServiceFactory, the ServiceFactory object represents an object that provides life cycle management for the BusinessService objects. The ServiceFactory object for enterprise beans is an EJBHome object.

Creating the Cache Class.

```
import java.util.ArrayList;
import java.util.List;

public class Cache
{
    private List<Service> services;

    public Cache()
    {
        services=new ArrayList<Service>();
    }

    public Service getService(String serviceName)
    {
        Service returnService=null;
        for(Service service:services)
```

```
        {
            if(serviceName.equalsIgnoreCase(service
            {
                System.out.println("Returning the C
                returnService=service;
                break;
            }
        }
        return returnService;
    }

    public void addService(Service newService)
    {
        boolean exists=false;
        for(Service service:services)
        {
            if(newService.getName().equalsIgnoreCas
            {
                exists=true;
            }
        }
        if(!exists)
        {
            services.add(newService);
        }
    }
}
```

▼ Step 5. Create the ServiceLocator Class, where the Service Locator abstracts the API lookup services, vendor dependencies, lookup complexities, and business object creation, and provides a simple interface to clients. This reduces the client's complexity. In addition, the same client or other clients can reuse the Service Locator.

Creating the ServiceLocator Class.

```
public class ServiceLocator
{
    public static Cache cache;

    static
    {
        cache=new Cache();
    }

    public static Service getService(String name)
    {
        Service service=cache.getService(name);
        if(service!=null)
        {
            return service;
        }

        InitialContext context=new InitialContext()
        Service serviceOne=(Service) context.lookup
        cache.addService(serviceOne);
        return serviceOne;
    }
}
```

▼ Step 6. Create the Runner Class, and validate the outputs.

Creating the ServiceLocatorPatternDemo Class

```
public class ServiceLocatorPatternDemo
{
    public static void main(String[] args)
    {
        Service service=ServiceLocator.getService('
        service.execute();
```

```
        service=ServiceLocator.getService("ServiceT
        service.execute();

        service=ServiceLocator.getService("ServiceO
        service.execute();

        service=ServiceLocator.getService("ServiceT
        service.execute();
    }
}
```

Output:

```
Creating a new ServiceOne Object
Executing ServiceOne
Creating a new ServiceTwo Object
Executing ServiceTwo
Returning the Cached ServiceOne object.
Executing ServiceOne
Returning the Cached ServiceTwo object.
Executing ServiceTwo
```

▼ Advantages

- Applications can optimize themselves at run-time by selectively adding and removing items from the service locator.
- Large sections of a library or application can be completely separated. The only link between them becomes the registry.

▼ Disadvantages

- The registry makes the code more difficult to maintain (opposed to using Dependency injection), because it becomes unclear when you would be introducing a breaking change.
- The registry hides the class dependencies causing run-time errors instead of compile-time errors when dependencies are

missing.

▼ Strategies

The following strategies are used to implement service Locator Pattern :

▼ EJB Service Locator Strategy

This strategy uses EJBHome object for enterprise bean components and this EJBHome is cached in the ServiceLocator for future use when the client needs the home object again.

▼ JMS Queue Service Locator Strategy

This strategy is applicable to point to point messaging requirements. The following the strategies under JMS Queue Service Locator Strategy.

- JMS Queue Service Locator Strategy
- JMS Topic Service Locator Strategy

▼ Type Checked Service Locator Strategy

This strategy has trade-offs. It reduces the flexibility of lookup, which is in the Services Property Locator strategy, but add the type checking of passing in a constant to the ServiceLocator.getHome() method.

▼ Session Facade Pattern

The Session Facade pattern’s core application is development of enterprise apps. You can also call it a logical extension of GoF designs. The pattern encases the interactions which are happening between the low-level components, which is Entity EJB. It is implemented as a higher level component, Session EJB. After which, it is responsible for providing a common and an only interface in order for the app to function or a part of the app.

It also reduces or completely ends the coupling between the lower level components which in turn simplifies the design. The structure of session facade pattern is such that it has client object, session facade object,

and business object. All of these have certain problems to cater and certain tasks to perform.

▼ Transfer Object Pattern

It is used when we want to pass data with multiple attributes in one shot from client to server. Transfer Object is a simple POJO class having getter/setter methods and is serialized so that it can be transferred over the network. Server Side business class normally fetches data from the database and fills the POJO and sends it to the client or passes it by value. For clients, the transfer object is read-only. The client can create its own transfer object and pass it to the server to update values in the database in one shot.

▼ Implementation

▼ Step 1. Create the Transfer Object, where it a Simple POJO having methods to set/get attributes only.

Creating the StudentTransferObject Class.

```
public class StudentTransferObject
{
    private String name;
    private int rollNo;

    public StudentTransferObject(String name,int ro
    {
        this.name=name;
        this.rollNo=rollNo;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
```

```
        this.name = name;
    }

    public int getRollNo()
    {
        return rollNo;
    }

    public void setRollNo(int rollNo)
    {
        this.rollNo = rollNo;
    }

    @Override
    public String toString()
    {
        return "StudentTransferObject [name=" + nam
    }

}
```

▼ Step 2. Create the Business Object. Which fills the Transfer Object with data.

Creating the StudentBusinessObject Class.

```
import java.util.ArrayList;
import java.util.List;

public class StudentBusinessObject
{
    List<StudentTransferObject> students;

    public StudentBusinessObject()
    {
        this.students=new ArrayList<StudentTransfer
```

```
        StudentTransferObject student1=new StudentT
        StudentTransferObject student2=new StudentT

        students.add(student1);
        students.add(student2);
    }

    public void deleteStudent(StudentTransferObject
    {
        students.remove(student);
        System.out.println("Student Deleted:"+stude
    }

    public List<StudentTransferObject> getAllStuden
    {
        return this.students;
    }

    public StudentTransferObject getStudent(int rol
    {
        return this.students.get(rollNo);
    }

    public void updateStudent(StudentTransferObject
    {
        students.get(student.getRollNo()).setName(s
        System.out.println("Updated Student:"+stude
    }
}
```

▼ Step 3. Create the Runner Class

Creating the TransferObjectDesignPattern Class

```
public class TransferObjectDesignPattern
{
    public static void main(String[] args)
    {
        StudentBusinessObject studentBusinessObject

        System.out.println("All Students:-");
        for(StudentTransferObject student: studentB
        {
            System.out.println(student);
        }
        System.out.println();

        StudentTransferObject student=studentBusine

        student.setName("Sagnik");
        studentBusinessObject.updateStudent(student

        student=studentBusinessObject.getStudent(0)
        System.out.println(student);
    }
}
```

Output:

```
All Students:-
StudentTransferObject [name=Adipta, rollNo=0]
StudentTransferObject [name=Aritra, rollNo=1]

Updated Student:StudentTransferObject [name=Sagnik,
rollNo=0]
StudentTransferObject [name=Sagnik, rollNo=0]
```

▼ Integration Layer Design Pattern

▼ Data Access Object Pattern

Data Access Object Pattern or DAO pattern is used to separate low-level data accessing API or operations from high-level business services. Following are the participants in Data Access Object Pattern.

▼ Implementation

▼ Step 1. Create the TransferObject, this represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Creating the Developer Class.

```
public class Developer
{
    private String name;
    private int developerId;

    public Developer(String name,int developerId)
    {
        this.name=name;
        this.developerId=developerId;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public int getDeveloperId()
```

```
    {
        return developerId;
    }

    public void setDeveloperId(int developerId)
    {
        this.developerId = developerId;
    }

    @Override
    public String toString()
    {
        return "Developer [name=" + name + ", devel
    }
}
```

▼ Step 2. Create the DataAccessObject level abstraction. The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source.

Creating the DeveloperDao Class.

```
import java.util.List;

public interface DeveloperDao
{
    public List<Developer> getAllDevelopers();
    public Developer getDeveloper(int developerId);
    public void updateDeveloper(Developer developer
    public void deleteDeveloper(Developer developer
}
```

▼ Step 3. Create the BusinessObject, that implements the DataAccessObject. The BusinessObject represents the data client. It

is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object in addition to a servlet or helper bean that accesses the data source.

Creating the DeveloperDaoImpl Class.

```
import java.util.ArrayList;
import java.util.List;

public class DeveloperDaoImpl implements DeveloperD
{
    List<Developer> developers;

    public DeveloperDaoImpl()
    {
        developers=new ArrayList<Developer>();

        Developer developer1=new Developer("Adipta'
        Developer developer2=new Developer("Aritra'

        developers.add(developer1);
        developers.add(developer2);
    }

    @Override
    public List<Developer> getAllDevelopers()
    {
        return this.developers;
    }

    @Override
    public Developer getDeveloper(int developerId)
    {
        Developer returndDeveloper=null;
        for(Developer developer:this.developers)
```

```
        {
            if(developerId==developer.getDeveloperI
            {
                returndDeveloper=developer;
                break;
            }
        }
        return returndDeveloper;
    }

    @Override
    public void updateDeveloper(Developer developer
    {
        Developer obtainedDeveloper=getDeveloper(de
        if(obtainedDeveloper!=null)
        {
            obtainedDeveloper.setName(developer.get
        }
        System.out.println("Updated Developer:"+obt
    }

    @Override
    public void deleteDeveloper(Developer developer
    {
        Developer obtainedDeveloper=getDeveloper(de
        if(obtainedDeveloper!=null)
        {
            this.developers.remove(obtainedDevelope
        }
        System.out.println("Deleted Developer:"+obt
    }
}
```

▼ Step 4. Create the Runner Class, that checks the implementation.

Creating the DataAccessObjectPatternDemo Class.

```
public class DataAccessObjectPatternDemo
{
    public static void main(String[] args)
    {
        DeveloperDao developerDao=new DeveloperDaoI

        System.out.println("Getting all Developers:
        for(Developer developer:developerDao.getAll
        {
            System.out.println(developer);
        }
        System.out.println();

        Developer developer=developerDao.getDevelop

        developer.setName("Sagnik");
        developerDao.updateDeveloper(developer);

        developer=developerDao.getDeveloper(0);
        System.out.println(developer);
    }
}
```

Output:

```
Getting all Developers:
Developer [name=Adipta, developerId=0]
Developer [name=Aritra, developerId=1]

Updated Developer:Developer [name=Sagnik,
developerId=0]
Developer [name=Sagnik, developerId=0]
```

▼ Advantages

1. The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently.
2. If we need to change the underlying persistence mechanism we only have to change the DAO layer and not all the places in the domain logic where the DAO layer is used from.

▼ Disadvantages

1. Potential disadvantages of using DAO is a leaky abstraction, code duplication, and abstraction inversion.

▼ Web Service Broker Pattern

The web service broker uses web protocols and XML. We can use this pattern to expose and broker the services. Assume a circumstance, where multiple organizations are lined up in order to request info from a number of service providers. A broker provides the central medium which makes the transfer of information happen.

It is a general gateway or an address in order for the client apps to be able to access a large, diverse variety of services. These services might need a broker in order to make an interaction with either only a single server app or multiple server apps. The broker performs certain tasks. It is responsible for receiving SOAP requests from the client apps in XML format along with authenticating the request and checking for the authorization. You can also use it to generate calls to multiple server apps, which depends on nature of the request.