Module 2: Design Patterns

Module Overview

In this module, students will be able to learn about the design patterns in java such as creational, structural, and behavioral design patterns.

Module Objective

After completion of this session, learners will be able to:

- understand the different design patterns that are the solutions for solving a specific problem/task.
- implement the different design patterns

Design Patterns

Introduction to Design Patterns

Design patterns are well-proved solution for solving a specific problem/task. Design patterns are programming language independent strategies for solving the common object-oriented design problems. That means a design pattern represents an idea, not a particular implementation.

Example:

- Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.
- Singleton design pattern is the best solution for the above specific problem. So, every design pattern has some specification or set of rules for solving the problems. What are those specifications, you will see later in the types of design patterns.

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

Creational design patterns

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

Types of creational design patterns

1. **Factory Method Pattern**

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as Virtual Constructor.

Advantages of Factory Design Pattern

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class so that it will work with any classes that implement that interface or that extend that abstract class.

Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

Calculate Electricity Bill: A Real-World Example of Factory Method

Step 1: Create a Plan abstract class.

```
import java.io.*;
abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}//end of Plan class.
```

Step 2: Create the concrete classes that extends Plan abstract class.

```
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
            rate=3.50;
       }
   }//end of DomesticPlan class.
```

```
class  CommercialPlan extends Plan{
   //@override
    public void getRate(){
       rate=7.50;
   }
/end of CommercialPlan class.
```

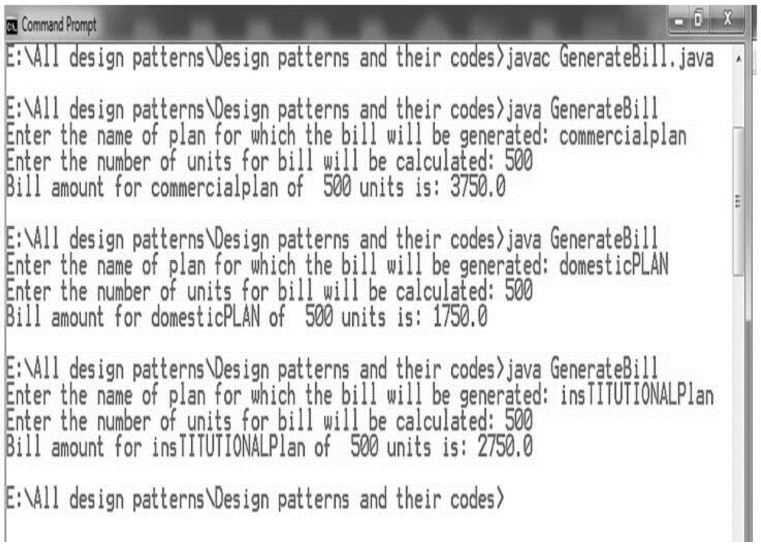
```
class  InstitutionalPlan extends Plan{
   //@override
    public void getRate(){
       rate=5.50;
   }
/end of InstitutionalPlan class.
```

Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information.

```
class GetPlanFactory{

  //use getPlan method to get object of type Plan
      public Plan getPlan(String planType){
           if(planType == null){
            return null;
           }
         if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
                return new DomesticPlan();
              }
          else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
               return new CommercialPlan();
           }
         else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
               return new InstitutionalPlan();
         }
     return null;
  }
}//end of GetPlanFactory class.
```

```
import java.io.*;
class GenerateBill{
   public static void main(String args[])throws IOException{
     GetPlanFactory planFactory = new GetPlanFactory();

     System.out.print("Enter the name of plan for which the bill will be generated: ");
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

     String planName=br.readLine();
     System.out.print("Enter the number of units for bill will be calculated: ");
     int units=Integer.parseInt(br.readLine());

     Plan p = planFactory.getPlan(planName);
     //call getRate() method and calculateBill()method of DomesticPaln.

      System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
          p.getRate();
          p.calculateBill(units);
           }
   }//end of GenerateBill class.
```

Output:



```
E:\All design patterns\Design patterns and their codes>javac GenerateBill.java

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: commercialplan
Enter the number of units for bill will be calculated: 500
Bill amount for commercialplan of  500 units is: 3750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: domesticPLAN
Enter the number of units for bill will be calculated: 500
Bill amount for domesticPLAN of  500 units is: 1750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: insTITUTIONALPlan
Enter the number of units for bill will be calculated: 500
Bill amount for insTITUTIONALPlan of  500 units is: 2750.0

E:\All design patterns\Design patterns and their codes>
```

2. Singleton design pattern

Singleton Pattern says that just "define a class that has only one instance and provides a global point of access to it" In other words, a class must ensure that only a single instance should be created and a single object can be used by all other classes.

There are two forms of singleton design pattern

- Early Instantiation: the creation of an instance at load time.
- Lazy Instantiation: the creation of instances when required.

Advantages of Singleton design pattern

- Saves memory because an object is not created at each request. Only a single instance is reused again and again.

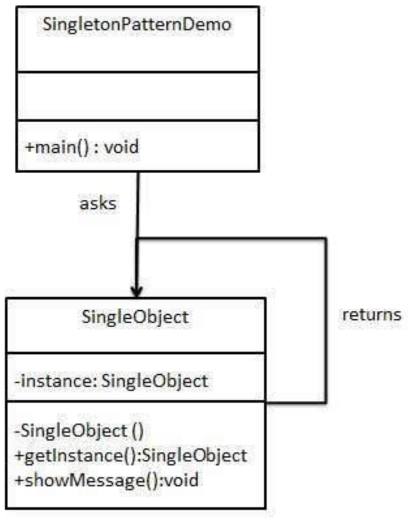
Usage of Singleton design pattern

- Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings, etc.

Implementation

We're going to create a SingleObject class. SingleObject class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.



Step 1: Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

  //create an object of SingleObject
  private static SingleObject instance = new SingleObject();

  //make the constructor private so that this class cannot be
  //instantiated
  private SingleObject(){}

  //Get the only object available
  public static SingleObject getInstance(){
    return instance;
  }

  public void showMessage(){
    System.out.println("Hello World!");
  }
}
```

Step 2: Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
  public static void main(String[] args) {

    //illegal construct
    //Compile Time Error: The constructor SingleObject() is not visible
    //SingleObject object = new SingleObject();

    //Get the only object available
    SingleObject object = SingleObject.getInstance();

    //show the message
    object.showMessage();
  }
}
```

Step 3: Verify the output.

```
Hello World!
```

3. Prototype design pattern

Prototype Pattern says that cloning an existing object instead of creating a new and can also be customized as per the requirement.

This pattern should be followed if the cost of creating a new object is expensive and resource-intensive.

Advantages of Prototype Pattern

The main advantages of the prototype pattern are as follows:

- It reduces the need for sub-classing.
- It hides the complexities of creating objects.
- The clients can get new objects without knowing which type of object they will be.
- It lets you add or remove objects at runtime.

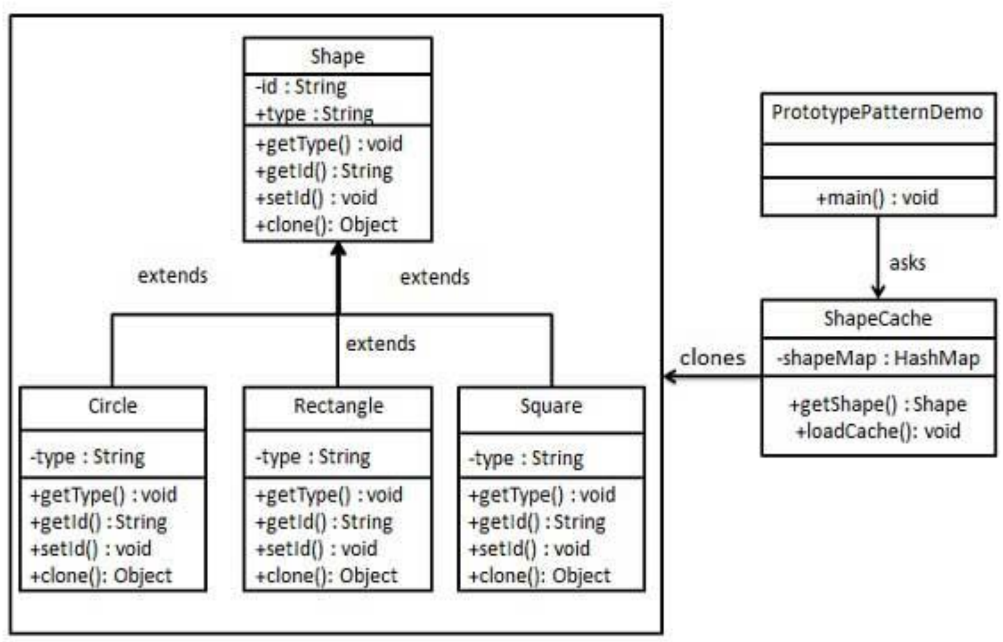
Usage of Prototype Pattern

- When the classes are instantiated at runtime.
- When the cost of creating an object is expensive or complicated.
- When you want to keep the number of classes in an application minimum.
- When the client application needs to be unaware of object creation and representation.

Implementation

We're going to create an abstract class Shape and concrete classes extending the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested.

PrototypPatternDemo, our demo class will use ShapeCache class to get a Shape object.



Step 1: Create an abstract class implementing Clonable interface.

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Step 2: Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape {
  public Rectangle(){
    type = "Rectangle";
  }

  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}
```

Square.java

```
public class Square extends Shape {
  public Square(){
    type = "Square";
  }

  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}
```

Circle.java

```
public class Circle extends Shape {
  public Circle(){
    type = "Circle";
  }

  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

Step 3: Create a class to get concrete classes from database and store them in a Hashtable.

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

Step 4: PrototypePatternDemo uses ShapeCache class to get clones of shapes stored in a Hashtable.

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
  public static void main(String[] args) {
    ShapeCache.loadCache();

    Shape clonedShape = (Shape) ShapeCache.getShape("1");
    System.out.println("Shape : " + clonedShape.getType());

    Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
    System.out.println("Shape : " + clonedShape2.getType());

    Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
    System.out.println("Shape : " + clonedShape3.getType());
  }
}
```

Step 5: Verify the output.

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

Structural design patterns

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplify the structure by identifying the relationships. These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

Types of structural design patterns

1. Decorator Pattern

A Decorator Pattern says that just "attach flexible additional responsibilities to an object dynamically".
In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.
The Decorator Pattern is also known as Wrapper.

Advantages of Decorator Pattern

- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object because changes are made by coding new classes.
- It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage of Decorator Pattern

It is used:

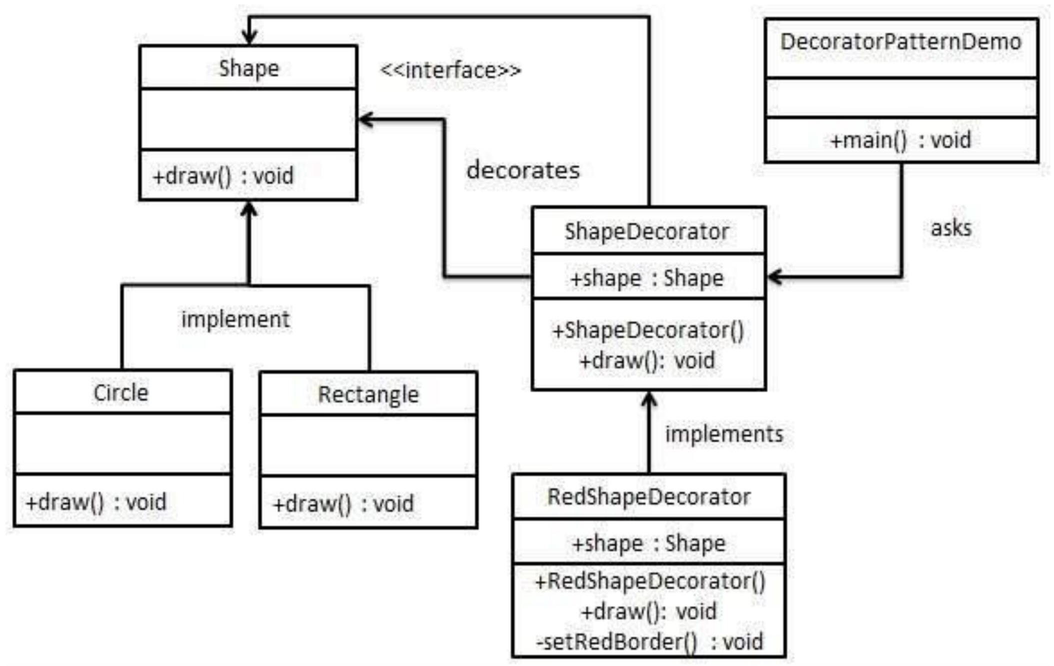
- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in the future.
- Extending functionality by sub-classing is no longer practical.

Implementation

We're going to create a Shape interface and concrete classes implementing the Shape interface. We will then create an abstract decorator class ShapeDecorator implementing the Shape interface and having Shape object as its instance variable.

RedShapeDecorator is concrete class implementing ShapeDecorator.

DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate Shape objects.



Step 1: Create an interface.

Shape.java

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

Step 2: Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Step 3: Create abstract decorator class implementing the Shape interface.

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
  protected Shape decoratedShape;

  public ShapeDecorator(Shape decoratedShape){
    this.decoratedShape = decoratedShape;
  }

  public void draw(){
    decoratedShape.draw();
  }
}
```

Step 4: Create concrete decorator class extending the ShapeDecorator class.

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

  public RedShapeDecorator(Shape decoratedShape) {
    super(decoratedShape);
  }

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

  private void setRedBorder(Shape decoratedShape){
    System.out.println("Border Color: Red");
  }
}
```

Step 5: Use the RedShapeDecorator to decorate Shape objects.

DecoratorPatternDemo.java

```
public class DecoratorPatternDemo {
  public static void main(String[] args) {

    Shape circle = new Circle();

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

    Shape redRectangle = new RedShapeDecorator(new Rectangle());
    System.out.println("Circle with normal border");
    circle.draw();

    System.out.println("\nCircle of red border");
    redCircle.draw();

    System.out.println("\nRectangle of red border");
    redRectangle.draw();
  }
}
```

Step 6: Verify the 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
```

2. Façade Pattern

A Facade Pattern says that just "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".

In other words, Facade Pattern describes a higher-level interface that makes the subsystem easier to use.

Practically, every Abstract Factory is a type of Facade.

Advantages of Facade Pattern

It shields the clients from the complexities of the sub-system components.

It promotes loose coupling between subsystems and their clients.

Usage of Facade Pattern

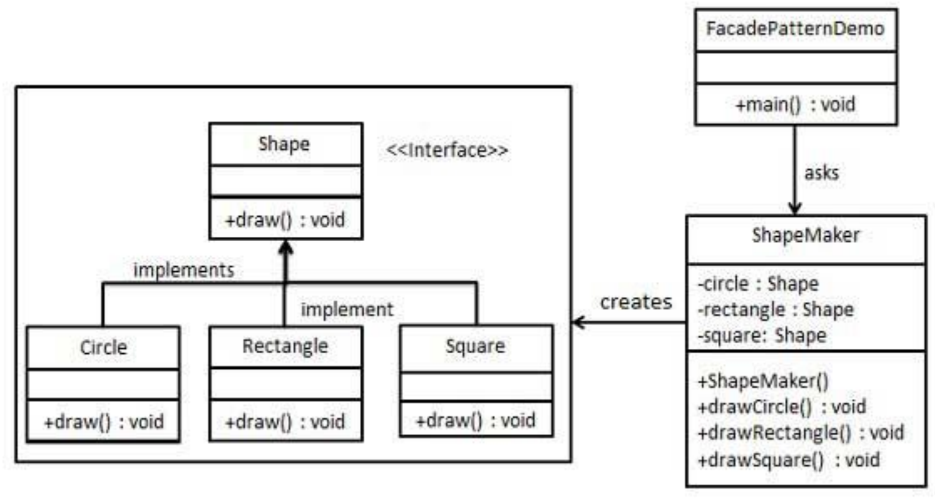
It is used:

- When you want to provide a simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.

Implementation

We are going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. FacadePatternDemo, our demo class, will use ShapeMaker class to show the results.



Step 1: Create an interface.

Shape.java

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

Step 2: Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

```
public class Square implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Step 3: Create a facade class.

ShapeMaker.java

```
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: Use the facade to draw various types of shapes.

FacadePatternDemo.java

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

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

Step 5: Verify the output.

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

Behavioral Design Pattern

Behavioral design patterns are concerned with the interaction and responsibility of objects.
In these design patterns, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.
That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

1. Chain of Responsibility Pattern

In a chain of responsibility design pattern, the sender sends a request to a chain of objects. The request can be handled by any object in the chain.
A Chain of Responsibility Pattern says that just "avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request". For example, an ATM uses the Chain of Responsibility design pattern in the money giving process.

In other words, we can say that normally each receiver contains a reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Advantages of Chain of Responsibility Pattern

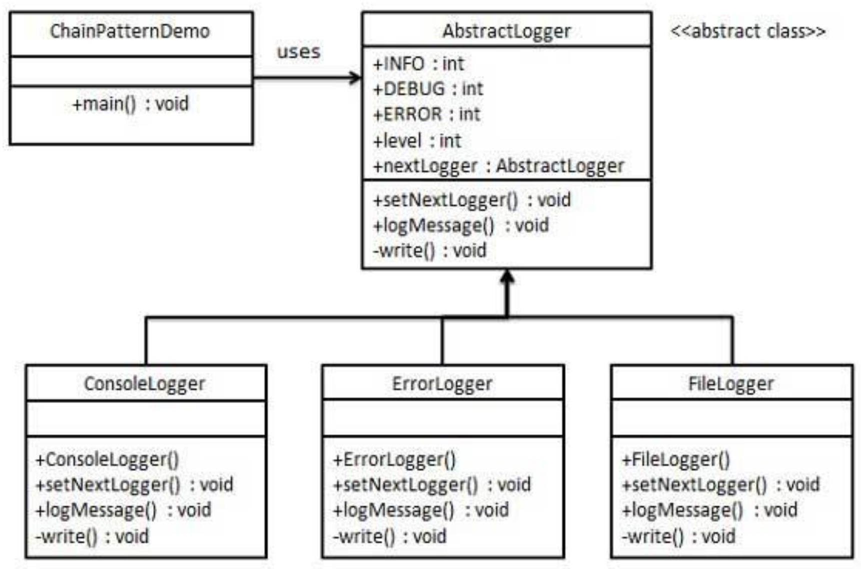
- It reduces the coupling.
- It adds flexibility while assigning responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Usage of Chain of Responsibility Pattern
It is used:

- When more than one object can handle a request and the handler is unknown.

- When the group of objects that can handle the request must be specified in a dynamic way.

Implementation
We have created an abstract class AbstractLogger with a level of logging. Then we have created three types of loggers extending the AbstractLogger. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Step 1: Create an abstract logger class.

AbstractLogger.java

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

  protected int level;

  //next element in chain or responsibility
  protected AbstractLogger nextLogger;

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

  public void logMessage(int level, String message){
    if(this.level <= level){
      write(message);
    }
    if(nextLogger !=null){
      nextLogger.logMessage(level, message);
    }
  }

  abstract protected void write(String message);

}
```

Step 2: Create concrete classes extending the logger.

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {

  public ConsoleLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Standard Console::Logger: " + message);
  }
}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {

  public ErrorLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Error Console::Logger: " + message);
  }
}
```

FileLogger.java

```
public class FileLogger extends AbstractLogger {

  public FileLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("File::Logger: " + message);
  }
}
```

Step 3: Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

ChainPatternDemo.java

```
public class ChainPatternDemo {

  private static AbstractLogger getChainOfLoggers(){

    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
    AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
    AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

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

    return errorLogger;
  }

  public static void main(String[] args) {
    AbstractLogger loggerChain = getChainOfLoggers();

    loggerChain.logMessage(AbstractLogger.INFO,
      "This is an information.");

    loggerChain.logMessage(AbstractLogger.DEBUG,
      "This is an debug level information.");

    loggerChain.logMessage(AbstractLogger.ERROR,
      "This is an error information.");
  }
}
```

Step 4: Verify the output.

```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

2. Iterator Pattern

Iterator Pattern is used "to access the elements of an aggregate object sequentially without exposing its underlying implementation".
The Iterator pattern is also known as Cursor.
In the collection framework, we are now using an Iterator that is preferred over Enumeration.
Advantage of Iterator Pattern

- It supports variations in the traversal of a collection.
- It simplifies the interface to the collection.

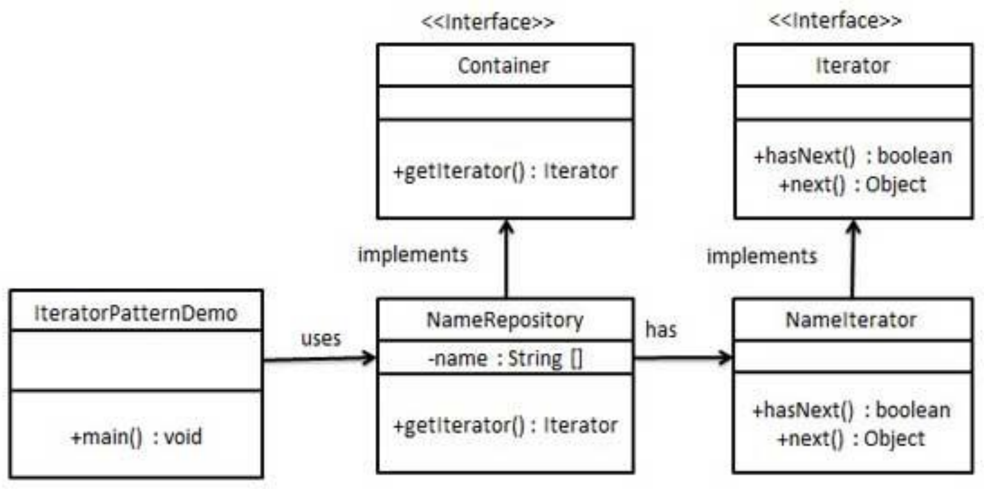
Usage of Iterator Pattern
It is used:

- When you want to access a collection of objects without exposing their internal representation.
- When there are multiple traversals of objects that need to be supported in the collection.

Implementation

We're going to create a Iterator interface which narrates navigation method and a Container interface which retruns the iterator . Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it

IteratorPatternDemo, our demo class will use NamesRepository, a concrete class implementation to print a Names stored as a collection in NamesRepository.



Step 1: Create interfaces.

Iterator.java

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

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Step 2: Create concrete class implementing the Container interface. This class has inner class NameIterator implementing the Iterator interface.

NameRepository.java

```
public class NameRepository implements Container {
  public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

  @Override
  public Iterator getIterator() {
    return new NameIterator();
  }

  private class NameIterator implements Iterator {

    int index;

    @Override
    public boolean hasNext() {

      if(index < names.length){
        return true;
      }
      return false;
    }

    @Override
    public Object next() {

      if(this.hasNext()){
        return names[index++];
      }
      return null;
    }
  }
}
```

Step 3: Use the NameRepository to get iterator and print names.

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Step 4: Verify the output.

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

Design Pattern in J2EE

J2EE design patterns are built for the development of the Enterprise Web-based Applications.
In J2EE, there are mainly three types of design patterns, which are further divided into their sub-parts:

- Presentation Layer Design Pattern
- Business Layer Design Pattern
- Integration Layer Design Pattern

Presentation Layer Design Pattern

1. Intercepting Filter Pattern

An Intercepting Filter Pattern says that "if you want to intercept and manipulate a request and response before and after the request is processed".

Usage:

- When you want centralization, common processing across requests, such as logging information about each request, compressing an outgoing response, or checking the data encoding scheme of each request.
- When you want pre and post-processing the components which are loosely coupled with core request-handling services to facilitate that are not suitable for addition and removal.

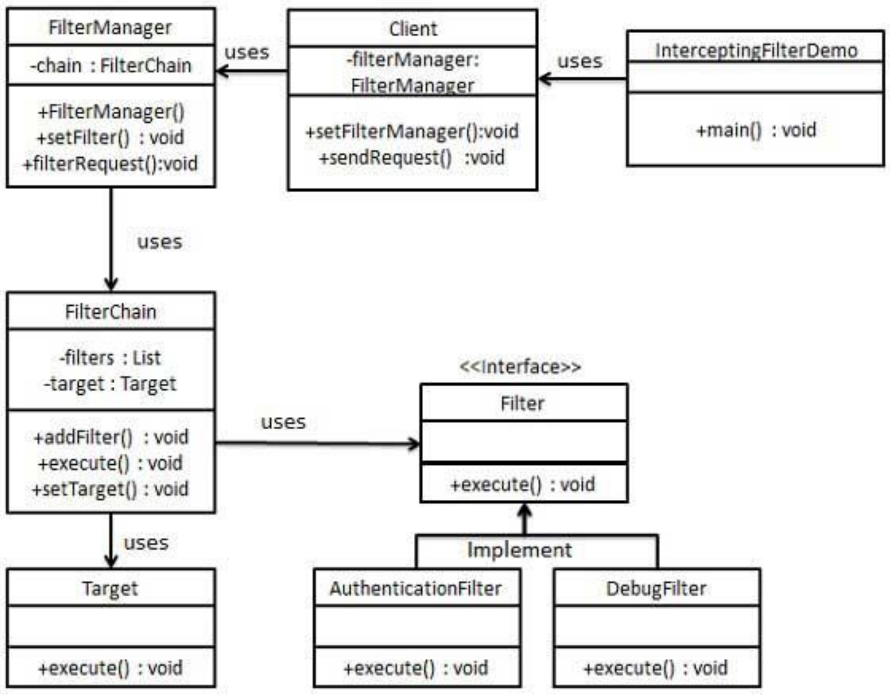
Benefits:

- It provides central control with loosely coupled handlers.
- It improves reusability.

Implementation

We are going to create a FilterChain,FilterManager, Target, Client as various objects representing our entities.AuthenticationFilter and DebugFilter represent concrete filters.

InterceptingFilterDemo, our demo class, will use Client to demonstrate Intercepting Filter Design Pattern.



Step 1: Create Filter interface.

Filter.java

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

Step 2: Create concrete filters.

AuthenticationFilter.java

```
public class AuthenticationFilter implements Filter {
   public void execute(String request){
      System.out.println("Authenticating request: " + request);
   }
}
```

DebugFilter.java

```
public class DebugFilter implements Filter {
  public void execute(String request){
    System.out.println("request log: " + request);
  }
}
```

Step 3: Create Target

Target.java

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

Step 4: Create Filter Chain

FilterChain.java

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

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

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

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

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

Step 5: Create Filter Manager

FilterManager.java

```
public class FilterManager {
   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 Client

Client.java

```
public class Client {
   FilterManager filterManager;

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

   public void sendRequest(String request){
      filterManager.filterRequest(request);
   }
}
```

Step 7: Use the Client to demonstrate Intercepting Filter Design Pattern.

InterceptingFilterDemo.java

```
public class InterceptingFilterDemo {
  public static void main(String[] args) {
    FilterManager filterManager = new FilterManager(new Target());
    filterManager.setFilter(new AuthenticationFilter());
    filterManager.setFilter(new DebugFilter());

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

Step 8: Verify the output.

```
Authenticating request: HOME
request log: HOME
Executing request: HOME
```

2. Front Controller Pattern

A Front Controller Pattern says that if you want to provide the centralized request handling mechanism so that all the requests will be handled by a single handler". This handler can do the authentication or authorization or logging or tracking of requests and then pass the requests to corresponding handlers.

Usage:

- When you want to control the page flow and navigation.
- When you want to access and manage the data model.
- When you want to handle the business processing.

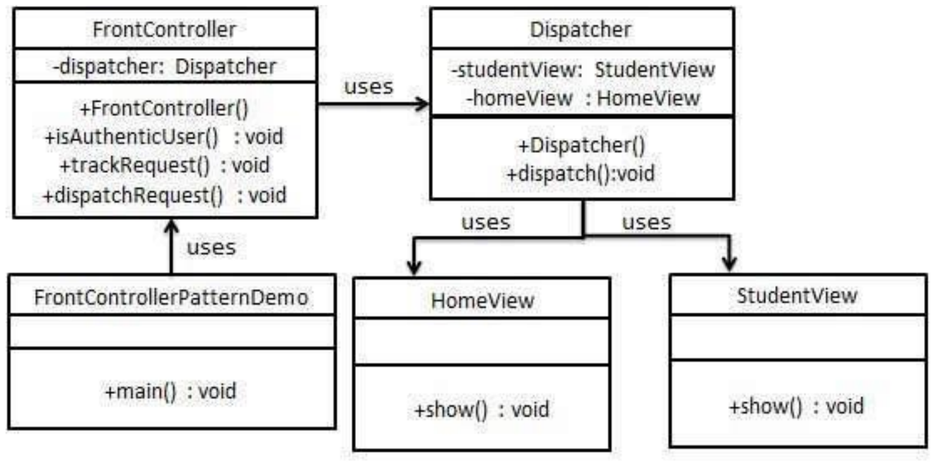
Benefits:

- It reduces the duplication of code in JSP pages, especially in those cases where several resources require the same processing.
- It maintains and controls a web application more effectively.

- A web application of two-tier architecture, the recommended approach is front controller to deal with user requests.

We are going to create a FrontController and Dispatcher to act as Front Controller and Dispatcher correspondingly. HomeView and StudentView represent various views for which requests can come to front controller.

FrontControllerPatternDemo, our demo class, will use FrontController to demonstrate Front Controller Design Pattern.



Step 1: Create Views.

HomeView.java

```
public class HomeView {
  public void show(){
    System.out.println("Displaying Home Page");
  }
```

StudentView.java

```
public class StudentView {
  public void show(){
    System.out.println("Displaying Student Page");
  }
```

Step 2: Create Dispatcher.

Dispatcher.java

```
public class Dispatcher {
  private StudentView studentView;
  private HomeView homeView;

  public Dispatcher(){
    studentView = new StudentView();
    homeView = new HomeView();
  }

  public void dispatch(String request){
    if(request.equalsIgnoreCase("STUDENT")){
      studentView.show();
    }
    else{
      homeView.show();
    }
  }
}
```

Step 3: Create FrontController

FrontController.java

```
public class FrontController {

   private Dispatcher dispatcher;

   public FrontController(){
      dispatcher = new Dispatcher();
   }

   private boolean isAuthenticUser(){
      System.out.println("User is authenticated successfully.");
      return true;
   }

   private void trackRequest(String request){
      System.out.println("Page requested: " + request);
   }

   public void dispatchRequest(String request){
      //log each request
      trackRequest(request);

      //authenticate the user
      if(isAuthenticUser()){
         dispatcher.dispatch(request);
      }
   }
}
```

Step 4: Use the FrontController to demonstrate Front Controller Design Pattern.

FrontControllerPatternDemo.java

```
public class FrontControllerPatternDemo {
  public static void main(String[] args) {

    FrontController frontController = new FrontController();
    frontController.dispatchRequest("HOME");
    frontController.dispatchRequest("STUDENT");
  }
}
```

Step 5: Verify the output.

```
Page requested: HOME
User is authenticated successfully.
Displaying Home Page
Page requested: STUDENT
User is authenticated successfully.
Displaying Student Page
```

Business Layer Design Pattern

1. Business Delegate Pattern

The business Delegate Pattern is used to decouple the presentation tier and business tier. It is basically used to reduce communication or remote lookup functionality to business tier code in presentation tier code. In the business tier, we have the following entities.

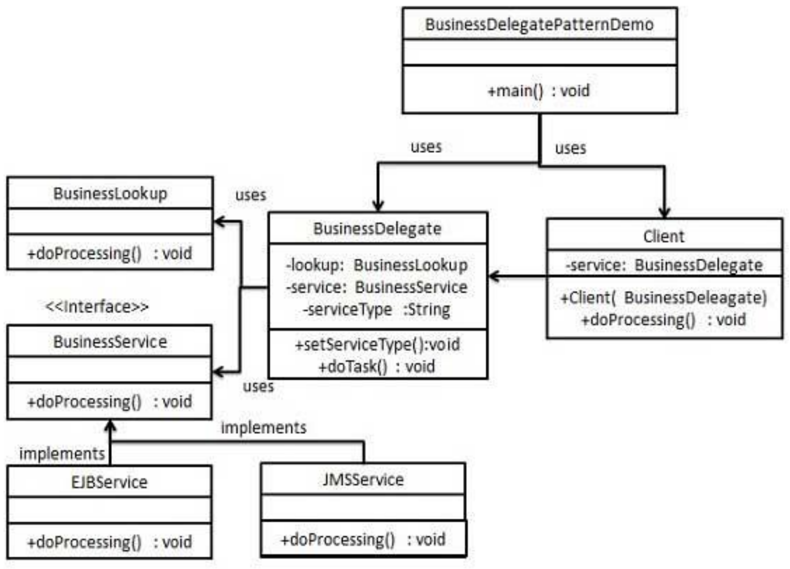
Client - Presentation tier code may be JSP, servlet, or UI java code.
Business Delegate - A single entry point class for client entities to provide access to Business Service methods.
LookUp Service – The lookup service object is responsible to get relative business implementation and provide business object access to the business delegate object.

Business Service - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

Implementation

We are going to create a Client, BusinessDelegate, BusinessService, LookUpService, JMSService and EJBService representing various entities of Business Delegate patterns.

BusinessDelegatePatternDemo, our demo class, will use BusinessDelegate and Client to demonstrate use of Business Delegate pattern.



Step 1: Create BusinessService Interface.

BusinessService.java

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

Step 2: Create concrete Service classes.

EJBService.java

```
public class EJBService implements BusinessService {

  @Override
  public void doProcessing() {
    System.out.println("Processing task by invoking EJB Service");
  }
}
```

JMSService.java

```
public class JMSService implements BusinessService {

  @Override
  public void doProcessing() {
    System.out.println("Processing task by invoking JMS Service");
  }
}
```

Step 3: Create Business Lookup Service.

BusinessLookUp.java

```
public class BusinessLookUp {
  public BusinessService getBusinessService(String serviceType){

    if(serviceType.equalsIgnoreCase("EJB")){
      return new EJBService();
    }
    else {
      return new JMSService();
    }
  }
```

Step 4: Create Business Delegate.

BusinessDelegate.java

```
public class BusinessDelegate {
   private BusinessLookUp lookupService = new BusinessLookUp();
   private BusinessService businessService;
   private String serviceType;

   public void setServiceType(String serviceType){
      this.serviceType = serviceType;
   }

   public void doTask(){
      businessService = lookupService.getBusinessService(serviceType);
      businessService.doProcessing();
   }
}
```

Step 5: Create Client.

Client.java

```
public class Client {

   BusinessDelegate businessService;

   public Client(BusinessDelegate businessService){
      this.businessService  = businessService;
   }

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

Step 6: Use BusinessDelegate and Client classes to demonstrate Business Delegate pattern.

BusinessDelegatePatternDemo.java

```
public class BusinessDelegatePatternDemo {

  public static void main(String[] args) {

    BusinessDelegate businessDelegate = new BusinessDelegate();
    businessDelegate.setServiceType("EJB");

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

    businessDelegate.setServiceType("JMS");
    client.doTask();
  }
}
```

Step 7: Verify the output.

```
Processing task by invoking EJB Service
Processing task by invoking JMS Service
```

2. Transfer Object Pattern

The Transfer Object pattern is used when we want to pass data with multiple attributes in one shot from client to server. A transfer object is also known as Value Object. Transfer Object is a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network. It does not have any behavior. 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 a client, the transfer object is read-only. A client can create its own transfer object and pass it to a server to update values in the database in one shot. Following are the entities of this type of design pattern.

Business Object - Business Service fills the Transfer Object with data.

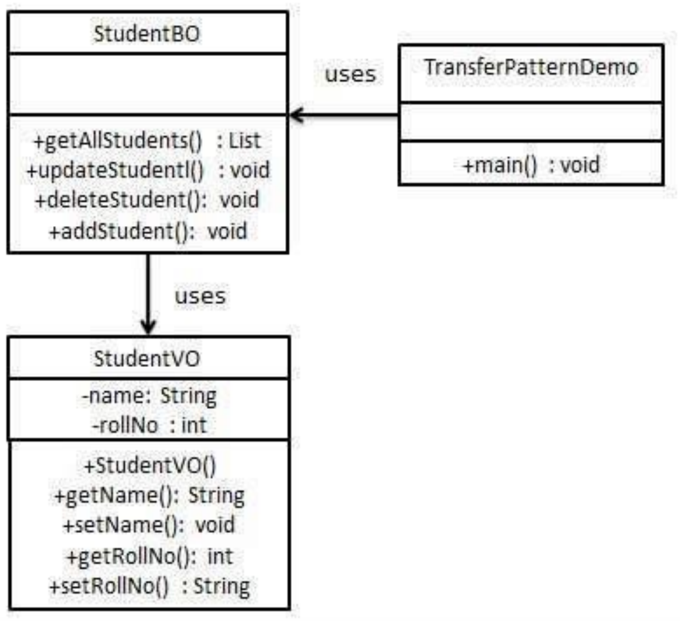
Transfer Object - Simple POJO having methods to set/get attributes only.

Client - Client either requests or sends the Transfer Object to Business Object.

Implementation

We are going to create a StudentBO as Business Object,Student as Transfer Object representing our entities.

TransferObjectPatternDemo, our demo class, is acting as a client here and will use StudentBO and Student to demonstrate Transfer Object Design Pattern.



Step 1: Create Transfer Object.

StudentVO.java

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

   StudentVO(String name, int rollNo){
      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;
   }
}
```

Step 2: Create Business Object.

StudentBO.java

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

public class StudentBO {

  //list is working as a database
  List<StudentVO> students;

  public StudentBO(){
    students = new ArrayList<StudentVO>();
    StudentVO student1 = new StudentVO("Robert",0);
    StudentVO student2 = new StudentVO("John",1);
    students.add(student1);
    students.add(student2);
  }
  public void deleteStudent(StudentVO student) {
    students.remove(student.getRollNo());
    System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");
  }

  //retrive list of students from the database
  public List<StudentVO> getAllStudents() {
    return students;
  }

  public StudentVO getStudent(int rollNo) {
    return students.get(rollNo);
  }

  public void updateStudent(StudentVO student) {
    students.get(student.getRollNo()).setName(student.getName());
    System.out.println("Student: Roll No " + student.getRollNo() +", updated in the database");
  }
}
```

Step 3: Use the StudentBO to demonstrate Transfer Object Design Pattern.

TransferObjectPatternDemo.java

```
public class TransferObjectPatternDemo {
  public static void main(String[] args) {
    StudentBO studentBusinessObject = new StudentBO();

    //print all students
    for (StudentVO student : studentBusinessObject.getAllStudents()) {
      System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
    }

    //update student
    StudentVO student = studentBusinessObject.getAllStudents().get(0);
    student.setName("Michael");
    studentBusinessObject.updateStudent(student);

    //get the student
    student = studentBusinessObject.getStudent(0);
    System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
  }
}
```

Step 4: Verify the output.

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```

Integration Layer Design Pattern

It refers to creating a messaging or data model that can be leveraged by consumers directly or indirectly. The data and/or message are then routed through an integration platform (e.g. Enterprise Service Bus) where they are then converted into a canonical standard format**.**

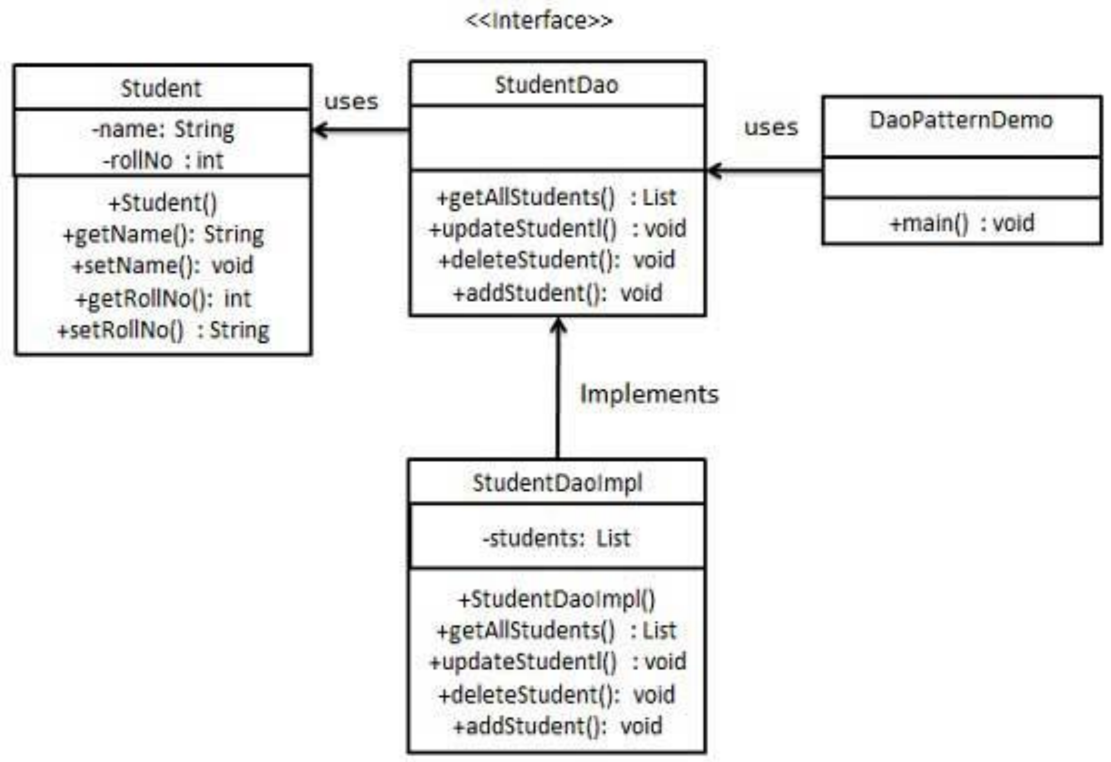
1. Data Access Object Pattern

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.

- **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
- **Data Access Object concrete class -** This class implements the above interface. This class is responsible to get data from a data source which can be a database / xml or any other storage mechanism.
- **Model Object or Value Object** - This object is a simple POJO containing get/set methods to store data retrieved using DAO class.

Implementation

We are going to create a Student object acting as a Model or Value Object.StudentDao is Data Access Object Interface.StudentDaoImpl is concrete class implementing Data Access Object Interface. DaoPatternDemo, our demo class, will use StudentDao to demonstrate the use of Data Access Object pattern



Step 1: Create Value Object.

Student.java

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

  Student(String name, int rollNo){
    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;
  }
}
```

Step 2
Create Data Access Object Interface.

StudentDao.java

```
import java.util.List;

public interface StudentDao {
  public List<Student> getAllStudents();
  public Student getStudent(int rollNo);
  public void updateStudent(Student student);
  public void deleteStudent(Student student);
}
```

Step 3: Create concrete class implementing above interface.

StudentDaoImpl.java

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

public class StudentDaoImpl implements StudentDao {

  //list is working as a database
  List<Student> students;

  public StudentDaoImpl(){
    students = new ArrayList<Student>();
    Student student1 = new Student("Robert",0);
    Student student2 = new Student("John",1);
    students.add(student1);
    students.add(student2);
  }
  @Override
  public void deleteStudent(Student student) {
    students.remove(student.getRollNo());
    System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");
  }

  //retrive list of students from the database
  @Override
  public List<Student> getAllStudents() {
    return students;
  }
```

Step 4: Use the StudentDao to demonstrate Data Access Object pattern usage.

DaoPatternDemo.java

```
public class DaoPatternDemo {
  public static void main(String[] args) {
    StudentDao studentDao = new StudentDaoImpl();

    //print all students
    for (Student student : studentDao.getAllStudents()) {
      System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
    }


    //update student
    Student student =studentDao.getAllStudents().get(0);
    student.setName("Michael");
    studentDao.updateStudent(student);

    //get the student
    studentDao.getStudent(0);
    System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

  }
}
```

Step 5: Verify the output.

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```

Exercise

Trainer will initiate a discussion of common questions on design pattern as given below:

1. **Which of the below is not a valid classification of design pattern?**

   a) Creational patterns b) Structural patterns
   c) Behavioural patterns d) Java Patterns

2. **Which design pattern provides a single class that provides simplified methods required by the client and delegates call to those methods?**

   a) Adapter pattern b) Builder pattern
   c) Facade pattern d) Prototype Pattern

3. **Which design pattern suggests multiple classes through which a request is passed and multiple but only relevant classes carry out operations on the request?**

   a) Singleton pattern b) Builder pattern
   c) Facade pattern d) Prototype Pattern

4. **Attach additional responsibilities to an object dynamically.It provides a flexible alternative to subclassing for extending functionality.**

   a) Chain of responsibility b) Adapter
   c) Decorator d) Prototype

5. **Which of the following pattern refers to creating duplicate object while keeping performance in mind?**

a) Builder pattern	b) Bridge pattern
c) Prototype pattern	d) Filter pattern