Lab Manual

Course: CSE-4122 (Object Oriented Design Pattern Lab)

Fourth Year, First Semester

Department of Computer Science and Engineering

The University of Rajaship

Table of Contents

Experiment No. 1: Creational Design: PatternsSingleton Pattern

1. Title

Implementation of the **1. Creational Design Patterns- Singleton Pattern** in Java.

2. Objectives

- To understand the **Creational Design Pattern** and its importance.
- To learn the **Singleton Pattern** and how it ensures only one instance of a class exists.
- To implement the Singleton Pattern in Java with proper access control and method calls.

3. Theory

- **Design Pattern**: A reusable solution to a commonly occurring problem in software design.
- **Creational Design Patterns**: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- **Singleton Pattern**:
    - Ensures that **only one instance** of a class is created throughout the application.
    - Provides a **global point of access** to that instance.
    - Commonly used in cases like logging, configuration settings, database connections, etc.

Key Properties of Singleton Pattern:

1. **Private Constructor** – prevents direct instantiation.
2. **Static Instance Variable** – holds the single instance of the class.
3. **Public Static Method** – provides global access to the instance.

4. Experiment Setup

We will implement two classes:

1. **SingleObject.java** – Contains the singleton implementation.
2. **SingletonPatternDemo.java** – Demonstrates how to use the singleton object.

5. Procedure

1. Create a Java class SingleObject with the following:
    - A **private constructor**.
    - A **private static instance variable**.
    - A **public static method** getInstance() that returns the single instance.
    - A method showMessage() to print a sample message.
2. Create another class SingletonPatternDemo:
    - Call the static method getInstance() to get the SingleObject.
    - Call the showMessage() method.
3. Compile and run the program.

6. UML Diagram:

```
+-------------------+
|     SingleObject       |
+-------------------+
| - instance               |    (private static)
| - SingleObject()    |    (private constructor)
+-------------------+
| + getInstance()      |    (public static)
| + showMessage()      |
+-------------------+
```

7. Program Code

SingleObject.java

```
// Singleton class
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 from Singleton Pattern!");       }}
```

SingletonPatternDemo.java

```
// Demo class
public class SingletonPatternDemo {
    public static void main(String[] args) {
        // Get the single object instance
        SingleObject object = SingleObject.getInstance();
        // Show message
        object.showMessage();       } }
```

9. Questions

1. What is the purpose of the Singleton Pattern?
2. Give two real-world examples where Singleton can be used.
3. Why is the constructor private in a singleton class?
4. How is Singleton different from a normal class?

Experiment No. 2: Creational Design: Factory Pattern

1. Title

Implementation of the **Factory Pattern** in Java.

2. Objectives

- To understand the concept of **Factory Design Pattern** under **Creational Design Patterns**.
- To learn how to create objects without exposing creation logic to the client.

- To implement a factory class that returns objects based on user input.

3. Theory

- **Factory Pattern**:
    - One of the most widely used creational patterns.
    - Defines an **interface** or an **abstract class** for creating an object, but lets subclasses decide which class to instantiate.
    - Helps in achieving **loose coupling** between client and implementation classes.

Key Features of Factory Pattern:

1. The client doesn't know the **exact implementation class** it is using.
2. Object creation logic is centralized in the **factory class**.
3. Promotes **code reusability** and **scalability**.

Real-world Analogy:

Think of a **shape factory**: You request a "circle" or "square" and the factory gives you the required shape object. You don't worry about how the shape object is created.

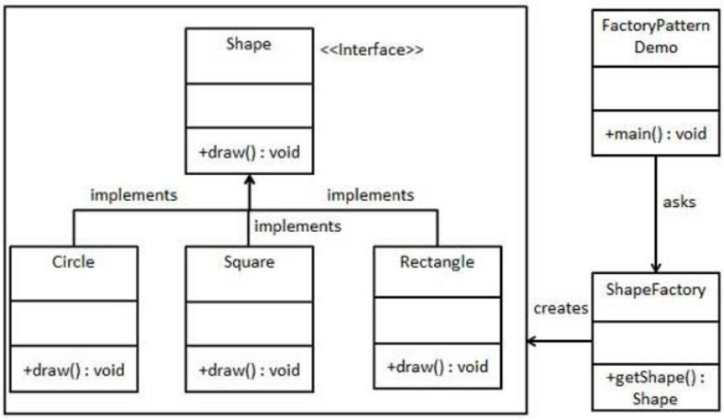
4. Experiment Setup

We will create the following:

1. **Shape.java** – Interface.
2. **Circle.java, Rectangle.java, Square.java** – Concrete classes implementing Shape.
3. **ShapeFactory.java** – Factory class that generates objects of concrete classes.
4. **FactoryPatternDemo.java** – Demo class to test the factory pattern.

5. Procedure

1. Define a Shape interface with a method draw().
2. Create Circle, Rectangle, and Square classes that implement the Shape interface.
3. Create a ShapeFactory class that has a method getShape(String shapeType) to return objects based on input.
4. Create a FactoryPatternDemo class that:
    - Calls ShapeFactory.
    - Requests a shape object (CIRCLE, RECTANGLE, SQUARE).
    - Calls the draw() method of that object.
5. Compile and run the program.
6. **UML Diagram**



Program Code

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

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

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

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

9. Questions

- What problem does the Factory Pattern solve?
- Why is the Factory Pattern preferred over directly creating objects with new?
- Can the Factory Pattern be extended to support more shapes without changing client code? How?
- Compare Factory Pattern and Singleton Pattern in terms of object creation.

```
ShapeFactory.java     // Factory class
public class ShapeFactory {
    // use getShape method to get object of type Shape
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;             }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();             } else if
(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();             } else if
(shapeType.equalsIgnoreCase("SQUARE")) {
```

```
FactoryPatternDemo.java     // Demo class
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();
        // Get an object of Rectangle and call its draw method
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();
        // Get an object of Square and call its draw method
```

Experiment No. 3 Creational Design: Abstract Pattern

1. Title

Implementation of the **Abstract Factory Pattern** in Java.

2. Objectives

- To understand the concept of the **Abstract Factory Design Pattern**.
- To learn how an **Abstract Factory (super-factory)** provides an interface to create families of related objects.
- To implement an abstract factory that returns factories of objects, instead of returning objects directly.

3. Theory

- **Abstract Factory Pattern**:
    - Also known as the **Factory of Factories**.
    - Provides an **interface** to create families of related objects without specifying their concrete classes.
    - Centralizes the creation logic of multiple factory classes.

Key Features:

1. Builds on the **Factory Pattern**, but adds a **layer of abstraction**.
2. Returns **factories** instead of direct objects.
3. Ensures the system is independent of how objects are created.

Real-world Analogy:

Imagine a **Factory Producer** that decides which factory to provide:

- **ShapeFactory** → creates different shapes.
- (Another factory like **ColorFactory** could create colors).
  The client requests through **FactoryProducer**, without worrying about actual implementation.

4. Experiment Setup

We will create the following:

1. **Shape.java** – Interface.
2. **Circle.java, Rectangle.java, Square.java** – Implementing Shape.
3. **AbstractFactory.java** – Abstract class that declares factory methods.
4. **ShapeFactory.java** – Concrete factory extending AbstractFactory.
5. **FactoryProducer.java** – Generates factories by passing information.
6. **AbstractFactoryPatternDemo.java** – Demo class to test the pattern.

5. Procedure

1. Create a Shape interface with draw() method.

2. Implement Circle, Rectangle, Square classes.
3. Create an AbstractFactory class with method getShape(String shapeType).
4. Create a ShapeFactory class extending AbstractFactory.
5. Create a FactoryProducer class with getFactory(String choice) method.
6. Create a demo class that:
    - Gets ShapeFactory from FactoryProducer.
    - Uses getShape() method to obtain required shape objects.
    - Calls the draw() method of these objects.

6.UML Diagarme

```
              +----------------------+
              |   AbstractFactory    |
              +----------------------+
              | + getShape(type)     |
              +----------------------+
                        ^
                        |
             +----------+----------+
             |                     |
  +------------------+   +------------------+
  |   ShapeFactory   |   | (OtherFactory)   |
  +------------------+   +------------------+
  | + getShape(type) |
  +------------------+

              +--------------------+
              |  FactoryProducer   |
              +--------------------+
              | + getFactory(type) |
              +--------------------+

     +--------------------+
     |       Shape        |
     +--------------------+
     | + draw()           |
     +--------------------+
       ^       ^       ^
```

```
         |          |          |
+--------+    +---+---+    +-------+
| Circle|    |Rectangle| |Square|
+--------+    +--------+  +-------+
| draw()|    | draw() | |draw()|
+--------+    +--------+  +-------+
```

7. Program Code

```
Shape.java// Shape interface
public interface Shape {
    void draw();
```

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

```
// Abstract Factory
public abstract class AbstractFactory {
    abstract Shape getShape(String shapeType);
}
```

```
// Concrete Factory extending AbstractFactory
public class ShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;           }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
```

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

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

```
// Factory producer class
public class FactoryProducer {
    public static AbstractFactory getFactory(String choice) {
        if (choice.equalsIgnoreCase("SHAPE")) {
            return new ShapeFactory();           }
        return null; // can extend for other factories like ColorFactory
    }}
```

```
// Demo class
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {
        // Get Shape Factory
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");
        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();
        // Get an object of Rectangle and call its draw method
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();
        // Get an object of Square and call its draw method
        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape3.draw();     }}
```

Experiment No. 4 Creational Design: Builder pattern

1. Title

Implementation of the **Builder Pattern** in Java.

2. Objectives

- To understand the **Builder Design Pattern** under **Creational Patterns**.
- To learn how to construct **complex objects step by step** using simpler objects.
- To implement a real-world example (fast-food restaurant meal builder) using the Builder Pattern.

3. Theory

- **Builder Pattern**:
  - Separates the construction of a **complex object** from its representation.
  - Allows the same construction process to create different representations.
  - Uses **composition of objects** rather than inheritance.

Key Features:

1. The **Builder** constructs the object step by step.
2. The object creation process is **independent** of the final object's parts.
3. Makes object creation flexible and easy to maintain.

Real-world Analogy (Fast-Food Meal):

- A **Meal** consists of multiple items like **Burgers** and **Cold Drinks**.
- Each item has a **packing** type (Wrapper for burgers, Bottle for drinks).
- The **MealBuilder** class assembles different combinations of burgers and drinks to form meals.

4. Experiment Setup

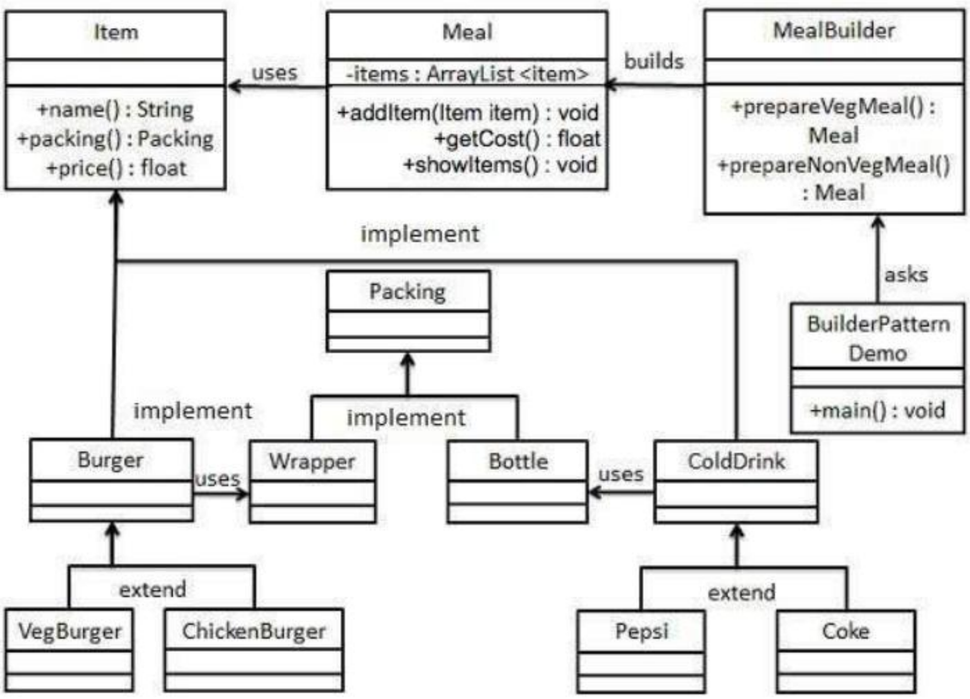
We will create the following:

1. **Packing.java** – Interface for packaging.
2. **Wrapper.java, Bottle.java** – Concrete classes implementing Packing.
3. **Item.java** – Interface for food items.
4. **Burger.java, ColdDrink.java** – Abstract classes implementing Item.
5. **VegBurger.java, ChickenBurger.java, Coke.java, Pepsi.java** – Concrete classes for items.
6. **Meal.java** – Class representing a meal (list of items).
7. **MealBuilder.java** – Builder class to build different meals.
8. **BuilderPatternDemo.java** – Demo class to run the program.

5. Procedure

1. Define a Packing interface with pack() method. Implement it using Wrapper and Bottle.
2. Define an Item interface with name(), packing(), and price() methods.
3. Create abstract classes Burger and ColdDrink implementing Item.
4. Implement concrete classes: VegBurger, ChickenBurger, Coke, and Pepsi.
5. Create a Meal class that:
    - Holds a list of Item.
    - Provides methods addItem(), getCost(), and showItems().
6. Create a MealBuilder class to build **Veg Meal** and **Non-Veg Meal**.
7. Create a demo class to build and display meals.

6.UML Diagram of Builder Pattern (Fast-Food Restaurant Example)




```
Wrapper.java    public class Wrapper implements Packing
{       @Override
        public String pack() {
                return "Wrapper";
        }
```

```
Bottle.java
  public class Bottle implements Packing {
        @Override
        public String pack() {
                return "Bottle";        }}}
```

8. Program Code

```
public interface Packing {
        String pack();}
```

```
Item.java
public interface Item {
        String name();
        Packing packing();
        float price();}
```

```
ColdDrink.java    public abstract class ColdDrink implements Item {
        @Override
        public Packing packing() {
                return new Bottle();          }
        @Override
        public abstract float price();}
```

```
VegBurger.java public class VegBurger extends Burger {
    @Override
    public float price() {
        return 25.0f;    }
    @Override    public String name() {
        return "Veg Burger";    }}
```

```
Burger.java public abstract class Burger implements Item {
    @Override
    public Packing packing() {
        return new Wrapper();     }
    @Override
    public abstract float price(); }
```

```
ChickenBurger.java     public class ChickenBurger extends Burger {
    @Override
    public float price() {
        return 50.5f;    }
    @Override    public String name() {
        return "Chicken Burger";    }}
```

```
// Concrete Factory extending AbstractFactory
public class ShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
```

```java
Coke.java public class Coke extends ColdDrink {
    @Override
    public float price() {
        return 30.0f;    }
    @Override    public String name() {        return "Coke";    }}
```

```java
Pepsi.java public class Pepsi extends ColdDrink {
    @Override    public float price() {
        return 35.0f;    }    @Override
    public String name() {        return "Pepsi";    }}
```

```java
Meal.java import java.util.ArrayList;
import java.util.List;
public class Meal {
    private List<Item> items = new ArrayList<Item>();
    public void addItem(Item item) {
        items.add(item);    }
    public float getCost() {
        float cost = 0.0f;        for (Item item : items) {
            cost += item.price();        }
        return cost;    }
    public void showItems() {        for (Item item : items) {
            System.out.print("Item : " + item.name());
            System.out.print(", Packing : " + item.packing().pack());
            System.out.println(", Price : " + item.price());
        }    }}
```

```java
MealBuilder.java public class MealBuilder {
    public Meal prepareVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new VegBurger());
        meal.addItem(new Coke());
        return meal;    }
    public Meal prepareNonVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;    }}
```

```java
BuilderPatternDemo.java    public class BuilderPatternDemo {
    public static void main(String[] args) {
        MealBuilder mealBuilder = new MealBuilder();
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: " + vegMeal.getCost());
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("¥nNon-Veg Meal");
        nonVegMeal.showItems();
        System.out.println("Total Cost: " + nonVegMeal.getCost());
    }}    }}
```

Expected Output:

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

Question:

- What is the main advantage of the Builder Pattern over Factory Pattern?
- How does the Builder Pattern improve object construction flexibility?
- Why is the packing logic separated in this example?
- Give another real-world system where Builder Pattern is useful.

Link: https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

https://refactoring.guru/design-patterns/abstract-factory

Experiment No. 5 Structural Design Patterns: •Adapter Pattern

Experiment Title

Implementation of Adapter Design Pattern in Java

Objective

- To understand the concept of the **Adapter Design Pattern**.
- To implement a program where an **audio player** (that can only play mp3) can also play vlc and mp4 formats by using an adapter.
- To demonstrate how the adapter acts as a **bridge** between incompatible interfaces.

Theory

- **Adapter Pattern** is a **structural design pattern** that allows objects with incompatible interfaces to work together.
- It converts the interface of one class into another interface expected by the client.

- Example: A **card reader** acts as an adapter between a memory card and a laptop.

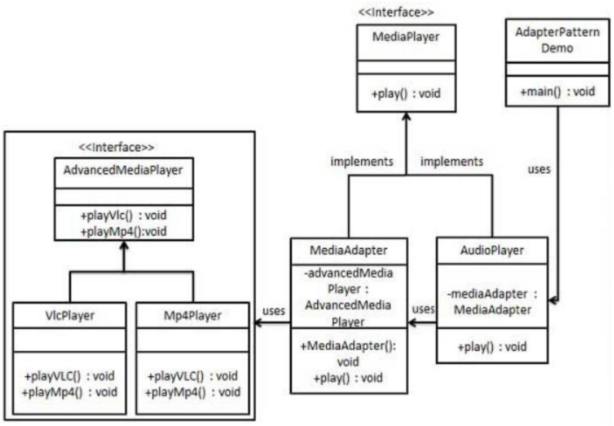
In this experiment:

- The AudioPlayer supports **mp3** files only.
- The AdvancedMediaPlayer supports **vlc** and **mp4**.
- The MediaAdapter is used to bridge the gap between AudioPlayer and AdvancedMediaPlayer.

Software & Hardware equirements

- **Software:** Java JDK 8 or later, any IDE (Eclipse, IntelliJ, NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

UML :



Step 1: Create the MediaPlayer interface	public interface MediaPlayer { void play(String audioType, String fileName);}

Step 2: Create the AdvancedMediaPlayer interface	public interface AdvancedMediaPlayer {
	void playVlc(String fileName);
	void playMp4(String fileName);}

Step 3: Implement VLC and MP4 players	public class VlcPlayer implements AdvancedMediaPlayer {
	@Override
	public void playVlc(String fileName) {
	System.out.println("Playing vlc file. Name: " + fileName); }
	@Override
	public void playMp4(String fileName) {
	// Do nothing }}
	public class Mp4Player implements AdvancedMediaPlayer {
	@Override
	public void playVlc(String fileName) {
	// Do nothing }

	@Override public void playMp4(String fileName) { System.out.println("Playing mp4 file. Name: " + fileName); } }

Step 4: Create the MediaAdapter class	public class MediaAdapter implements MediaPlayer {
	AdvancedMediaPlayer advancedMusicPlayer;
	public MediaAdapter(String audioType) {
	if(audioType.equalsIgnoreCase("vlc")) {
	advancedMusicPlayer = new VlcPlayer();
	} else if(audioType.equalsIgnoreCase("mp4")) {
	advancedMusicPlayer = new Mp4Player(); } }
	@Override
	public void play(String audioType, String fileName) {
	if(audioType.equalsIgnoreCase("vlc")) {
	advancedMusicPlayer.playVlc(fileName);
	} else if(audioType.equalsIgnoreCase("mp4")) {
	advncedMusicPlayer.playMp4(fileName); } }}

Step 5: Create the AudioPlayer class (client)	public class AudioPlayer implements MediaPlayer {
	MediaAdapter mediaAdapter;
	@Override
	public void play(String audioType, String fileName) {
	if(audioType.equalsIgnoreCase("mp3")) {
	System.out.println("Playing mp3 file. Name: " + fileName);
	} else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")) {
	mediaAdapter = new MediaAdapter(audioType);
	mediaAdapter.play(audioType, fileName); } else {
	System.out.println("Invalid media. " + audioType + " format not supported"); } }}

Step 6: Test the Adapter Pattern	public class AdapterPatternDemo {
	public static void main(String[] args) {
	AudioPlayer audioPlayer = new AudioPlayer();
	audioPlayer.play("mp3", "song1.mp3");
	audioPlayer.play("mp4", "movie1.mp4");
	audioPlayer.play("vlc", "video1.vlc");
	audioPlayer.play("avi", "clip1.avi"); }}

Expected Output:

Playing mp3 file. Name: song1.mp3

Playing mp4 file. Name: movie1.mp4

Playing vlc file. Name: video1.vlc

Invalid media. avi format not supported

Result

The experiment successfully demonstrates the **Adapter Pattern** by allowing the AudioPlayer (which can play only mp3 files) to also play vlc and mp4 files through the MediaAdapter.

Viva Questions

1. What is the Adapter Pattern and why is it used?
2. Is Adapter Pattern structural, behavioral, or creational? Why?
3. Give a real-life example of the Adapter Pattern.
4. How is Adapter different from Decorator and Bridge patterns?
5. Can Adapter Pattern be used in multiple inheritance scenarios?

Experiment No. 6 Structural Design Patterns: Bridge Pattern

Experiment Title

Implementation of Bridge Design Pattern in Java

Objective

- To understand the concept of the **Bridge Design Pattern**.
- To implement a program where the **abstraction (Shape)** is decoupled from its **implementation (Drawing API)**.
- To demonstrate how different implementations (red and green circles) can be interchanged without modifying the abstraction.

Theory

- **Bridge Pattern** is a **structural design pattern** that **decouples abstraction from its implementation**, allowing both to vary independently.
- It uses a **bridge interface** that connects an abstract class with implementation classes.
- Example: Consider a **remote control** (abstraction) and **TV/Radio** (implementations). The remote can control both without changing its structure.

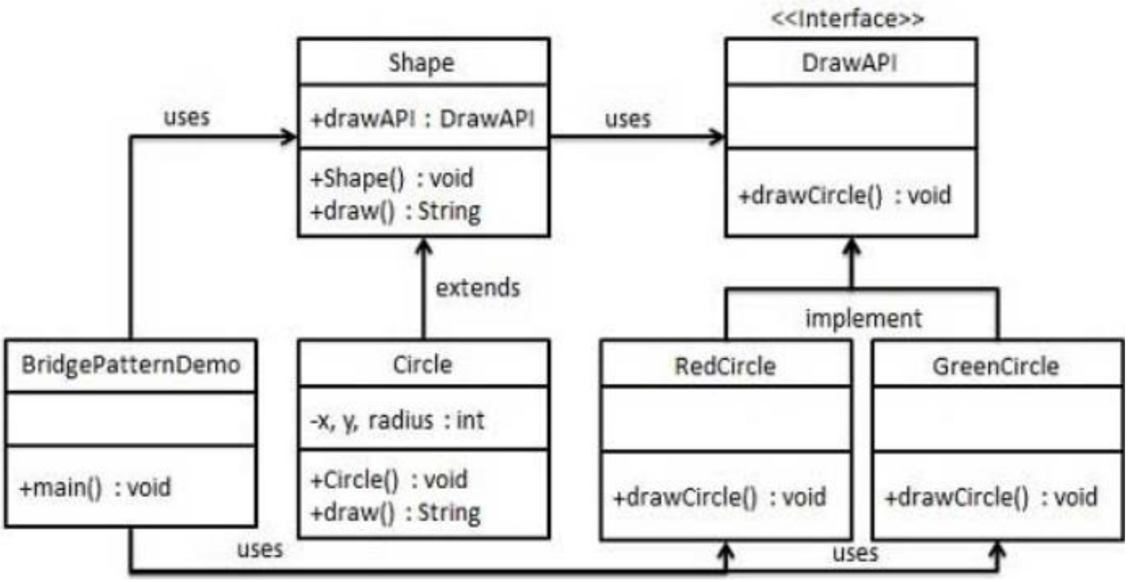
In this experiment:

- **Abstraction:** Shape (with reference to DrawAPI)
- **Refined Abstraction:** Circle
- **Implementor Interface:** DrawAPI
- **Concrete Implementors:** RedCircle, GreenCircle

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, or NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

UML:



Code:

Step 1: Create interface	**public interface DrawAPI {** **void drawCircle(int radius, int x, int y);** **}**

Step 2: Implement the concrete classes	**public class RedCircle implements DrawAPI {**
	    @Override
	    public void drawCircle(int radius, int x, int y) {
	        System.out.println("Drawing Circle [color: red, radius: " + radius + ", x: " + x + ", y:" + y + "]"); } }
	public class GreenCircle implements DrawAPI {
	    @Override
	    public void drawCircle(int radius, int x, int y) {
	        System.out.println("Drawing Circle [color: green, radius: " + radius + ", x: " + x + ", y:" + y +
	"]"); }}

Step 3: Create the abstract class Shape	**public abstract class Shape {**
	    protected DrawAPI drawAPI;
	    protected Shape(DrawAPI drawAPI) {
	        this.drawAPI = drawAPI; }
	    public abstract void draw(); // Bridge method}

Step 4: Implement the Circle class	**public class Circle extends Shape {**
	    private int x, y, radius;
	    public Circle(int x, int y, int radius, DrawAPI drawAPI) {

	super(drawAPI); this.x = x; this.y = y; this.radius = radius; } **@Override** **public void draw() { drawAPI.drawCircle(radius, x, y); }}**
Step 5: Test the Bridge Pattern	**public class BridgePatternDemo {** **public static void main(String[] args) {** **Shape redCircle = new Circle(100, 100, 10, new RedCircle());** **Shape greenCircle = new Circle(200, 200, 20, new GreenCircle()); redCircle.draw();** **greenCircle.draw(); }}**
Expected Output	**Drawing Circle [color: red, radius: 10, x: 100, y:100]** **Drawing Circle [color: green, radius: 20, x: 200, y:200]**

Result

The experiment successfully demonstrates the **Bridge Design Pattern** by decoupling the abstraction (Shape) from its implementations (DrawAPI). This allows the abstraction and implementation to vary independently.

Viva Questions

1. What is the Bridge Pattern and why is it used?
2. How does Bridge differ from Adapter Pattern?
3. Can the Bridge Pattern help in achieving loose coupling? How?
4. Give a real-life example of the Bridge Pattern.
5. What happens if we add a new shape or a new drawing implementation?

Experiment No. 7 Structural Design Patterns: Composite Pattern

Experiment Title

Implementation of Composite Design Pattern in Java

Objective

- To understand the concept of the **Composite Design Pattern**.
- To implement a program where an **organization's employee hierarchy** is represented using a tree structure.

- To demonstrate how a group of objects (employees) and individual objects can be treated uniformly.

Theory

- **Composite Pattern** is a **structural design pattern**.
- It is used when we need to treat a group of objects in the **same way** as a single object.
- It composes objects into **tree structures** to represent **part-whole hierarchies**.
- Example: An **organization chart** where the **CEO** has managers under them, and managers have employees.

In this experiment:

- **Leaf Nodes:** Employees without subordinates.
- **Composite Nodes:** Employees who have subordinates (e.g., managers).
- The same operations (like adding, removing, displaying details) can be applied to both.

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

Step 1: Create the Employee class

```
import java.util.ArrayList;
import java.util.List;
public class Employee {
    private String name;
    private String dept;
    private int salary;
    private List<Employee> subordinates;
    // Constructor
    public Employee(String name, String dept, int salary) {
        this.name = name;
        this.dept = dept;
        this.salary = salary;
        subordinates = new ArrayList<Employee>();    }
    public void add(Employee e) {
        subordinates.add(e);    }
    public void remove(Employee e) {
        subordinates.remove(e);    }
    public List<Employee> getSubordinates() {
        return subordinates;    }
    @Override
```

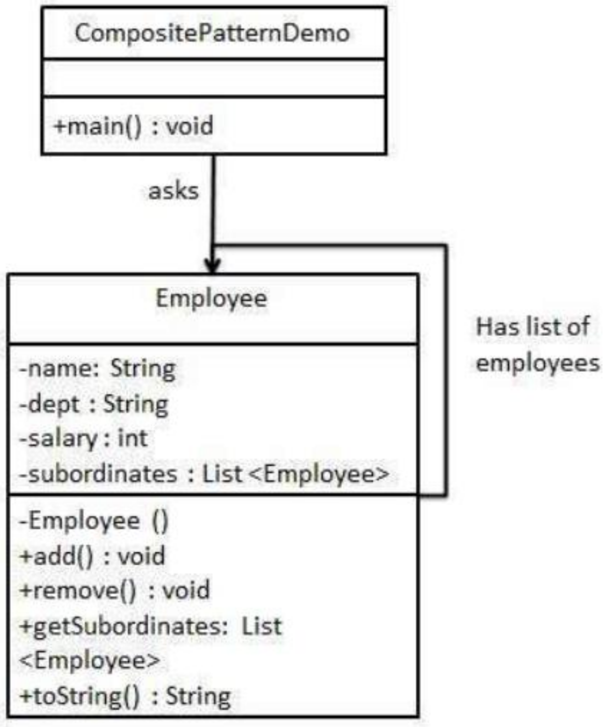
	public String toString() { **return ("Employee :[Name : " + name + ", dept : " + dept + ", salary :" + salary + "]"); }}**
Step 2: Test the Composite Pattern	**public class CompositePatternDemo {** **public static void main(String[] args) {** **Employee CEO = new Employee("John", "CEO", 30000);** **Employee headSales = new Employee("Robert", "Head Sales", 20000);** **Employee headMarketing = new Employee("Michel", "Head Marketing", 20000);** **Employee clerk1 = new Employee("Laura", "Marketing", 10000);** **Employee clerk2 = new Employee("Bob", "Marketing", 10000);** **Employee salesExecutive1 = new Employee("Richard", "Sales", 10000);** **Employee salesExecutive2 = new Employee("Rob", "Sales", 10000);** **CEO.add(headSales);** **CEO.add(headMarketing);** **headSales.add(salesExecutive1);** **headSales.add(salesExecutive2);** **headMarketing.add(clerk1);** **headMarketing.add(clerk2);** **// Print the organization hierarchy** **System.out.println(CEO);** **for (Employee headEmployee : CEO.getSubordinates()) {** **System.out.println(" " + headEmployee);** **for (Employee employee : headEmployee.getSubordinates()) {** **System.out.println(" " + employee); } } }}**
Expected Output	**Employee :[Name : John, dept : CEO, salary :30000]** **Employee :[Name : Robert, dept : Head Sales, salary :20000]** **Employee :[Name : Richard, dept : Sales, salary :10000]** **Employee :[Name : Rob, dept : Sales, salary :10000]** **Employee :[Name : Michel, dept : Head Marketing, salary :20000]** **Employee :[Name : Laura, dept : Marketing, salary :10000]** **Employee :[Name : Bob, dept : Marketing, salary :10000]**

Result

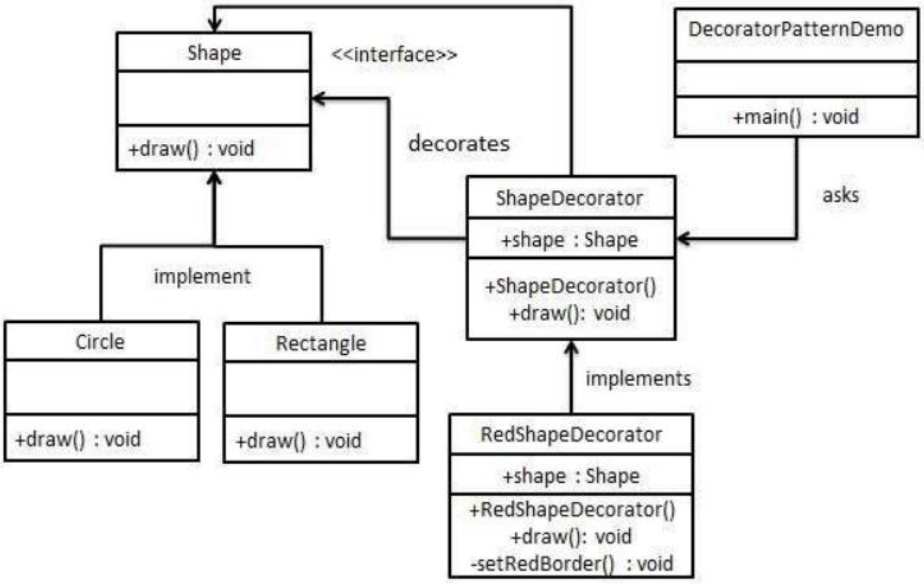
The experiment successfully demonstrates the **Composite Pattern** by creating an organization hierarchy where objects (Employee) can contain other objects of the same type, forming a tree structure.

Viva Questions

1. What is the Composite Pattern and why is it used?
2. How does the Composite Pattern represent part-whole hierarchies?
3. Give a real-life example of Composite Pattern other than organization hierarchy.
4. What is the difference between Composite Pattern and Decorator Pattern?
5. How can you extend this pattern to handle more complex hierarchies?



Experiment No. 8 Structural Design Patterns: Decorator pattern



Experiment Title

Implementation of Decorator Design Pattern in Java

Objective

- To understand the concept of the **Decorator Design Pattern**.
- To implement a program where a **shape** can be decorated with additional features (e.g., border color) without modifying the original shape class.
- To demonstrate how new functionality can be added dynamically at runtime by wrapping objects.

Theory

- **Decorator Pattern** is a **structural design pattern**.
- It allows behavior to be added to individual objects dynamically, without modifying their class.
- The decorator class **wraps** the original object and **provides additional functionality** while keeping the class methods intact.
- Example: A **Christmas tree** (base object) can be decorated with **lights, ornaments, and tinsel** dynamically.

In this experiment:

- **Component Interface:** Shape
- **Concrete Components:** Circle, Rectangle

- **Decorator (abstract class):** ShapeDecorator
- **Concrete Decorator:** RedShapeDecorator (adds border color functionality)

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, or NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

Code:

Step 1: Create interface	**public interface Shape {** **void draw(); }**
Step 2: Implement the concrete classes	**public class Rectangle implements Shape {** **@Override** **public void draw() { System.out.println("Shape: Rectangle"); }}** **public class Circle implements Shape {** **@Override** **public void draw() { System.out.println("Shape: Circle"); }}**
Step 3: Create the abstract decorator class	**public abstract class ShapeDecorator implements Shape {** **protected Shape decoratedShape;** **public ShapeDecorator(Shape decoratedShape) {** **this.decoratedShape = decoratedShape; }** **public void draw() {** **decoratedShape.draw(); }}**
Step 4: Create the concrete decorator	**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: Test the Decorator Pattern	**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 without border:"); circle.draw();**

	System.out.println("¥nCircle with red border:"); **redCircle.draw();** **System.out.println("¥nRectangle with red border:");** **redRectangle.draw(); }}**
Expected Output	**Circle without border:** **Shape: Circle** **Circle with red border:** **Shape: Circle** **Border Color: Red** **Rectangle with red border:** **Shape: Rectangle** **Border Color: Red**

Viva Questions

1. What is the Decorator Pattern and why is it used?
2. How does Decorator Pattern differ from Inheritance?
3. Give a real-life example of the Decorator Pattern.
4. Can we add multiple decorators to the same object? Explain.
5. What are the advantages of using the Decorator Pattern over subclassing?

Experiment No. 9 Behavioral Design Patterns: • Chain of Responsibility Pattern

Experiment Title

Implementation of Chain of Responsibility Design Pattern in Java

Objective

- To understand the **Chain of Responsibility Pattern**.
- To implement a program where multiple handler objects form a chain, and a request is passed along until one of them handles it.
- To demonstrate how this pattern **decouples the sender and receiver** of a request.

Theory

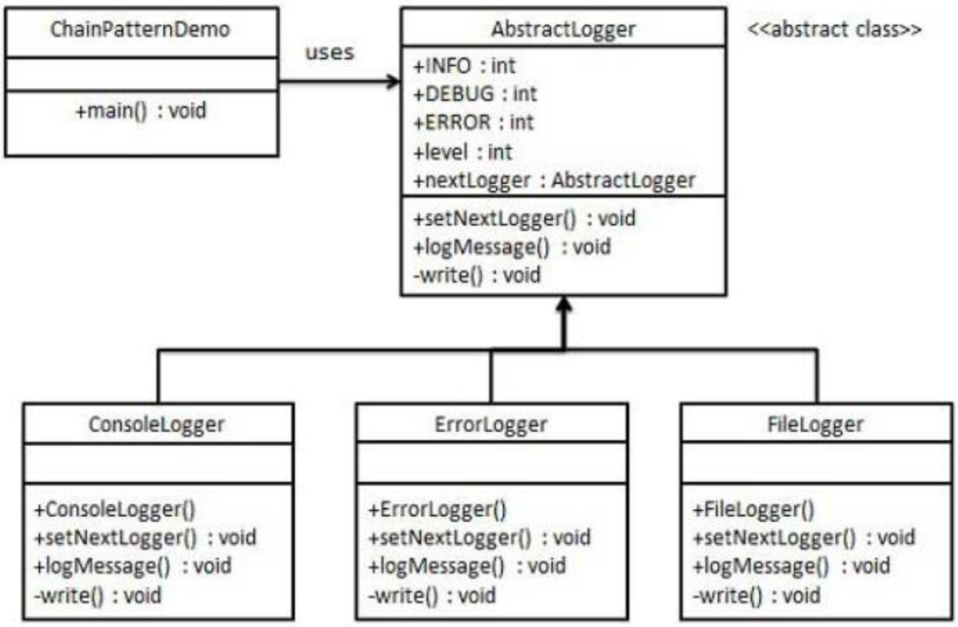
- The **Chain of Responsibility Pattern** is a **behavioral design pattern**.
- It creates a **chain of receiver objects** for handling requests.
- Each handler has:

- **Reference to the next handler** in the chain.
- **Logic to handle or forward the request**.

- If a handler can process the request, it does; otherwise, it passes it to the next handler in the chain.
- Real-life example:
    - A **support system** where requests are passed from **junior staff** → **supervisor** → **manager** depending on the complexity.

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, or NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.



Code:

Step 1: Create an abstract handler class	**abstract class Logger {** **public static int INFO = 1;** **public static int DEBUG = 2;** **public static int ERROR = 3;** **protected int level;** **protected Logger nextLogger;** **public void setNextLogger(Logger 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 logger classes	**class ConsoleLogger extends Logger {** **public ConsoleLogger(int level) {** **this.level = level; }** **@Override** **protected void write(String message) {** **System.out.println("Standard Console::Logger: " + message); }}** **class FileLogger extends Logger {** **public FileLogger(int level) {** **this.level = level; }** **@Override** **protected void write(String message) {** **System.out.println("File::Logger: " + message); }}** **class ErrorLogger extends Logger {** **public ErrorLogger(int level) {** **this.level = level; }** **@Override** **protected void write(String message) {** **System.out.println("Error Console::Logger: " + message); }}**

Step 3: Build the chain of responsibility	**public class ChainPatternDemo {**
	private static Logger getChainOfLoggers() {
	Logger errorLogger = new ErrorLogger(Logger.ERROR);
	Logger fileLogger = new FileLogger(Logger.DEBUG);
	Logger consoleLogger = new ConsoleLogger(Logger.INFO);
	errorLogger.setNextLogger(fileLogger);
	fileLogger.setNextLogger(consoleLogger);
	return errorLogger; // starting point }
	public static void main(String[] args) {
	Logger loggerChain = getChainOfLoggers();
	loggerChain.logMessage(Logger.INFO, "This is an information.");
	loggerChain.logMessage(Logger.DEBUG, "This is a debug level information.");
	loggerChain.logMessage(Logger.ERROR, "This is an error information."); }}

Expected Output	**iStandard Console::Logger: This is an information.** **File::Logger: This is a debug level information.**

	Standard Console::Logger: This is a 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.**

Result

The experiment successfully demonstrates the **Chain of Responsibility Pattern**, where requests are passed along a chain of handlers until they are processed.

Viva Questions

1. What is the Chain of Responsibility Pattern?
2. Which design pattern category does it belong to?
3. What is the main advantage of using this pattern?
4. How does it decouple sender and receiver?
5. Can you give a real-life example of this pattern?

Experiment No. 10 Behavioral Design Patterns: • Command Pattern

Experiment Title

Implementation of Command Design Pattern in Java

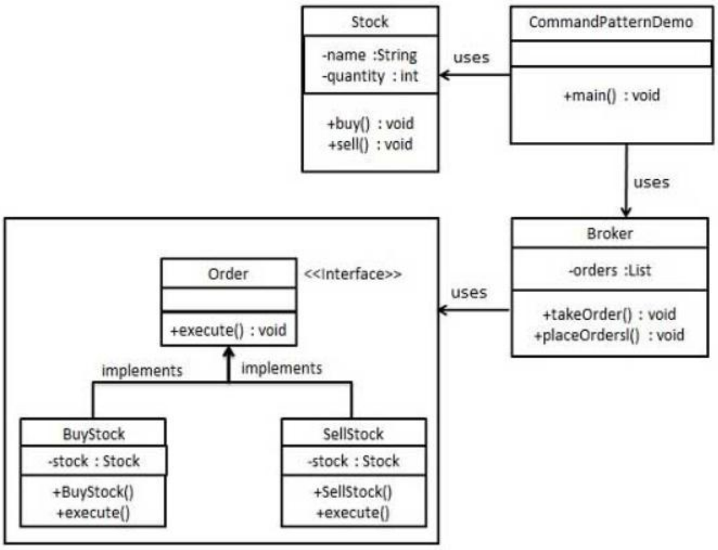
Objective

- To understand the concept of the **Command Design Pattern**.
- To implement a program where requests (commands) are encapsulated as objects.
- To demonstrate how a client can issue requests without knowing the exact receiver or implementation details.

Theory

- **Command Pattern** is a **behavioral design pattern**.
- It turns a request into a **standalone object (command)** that contains all the information about the request.

- The pattern involves **four key components**:



1. **Command Interface** – Declares the execution method.
2. **Concrete Command** – Implements the command by binding a receiver.
3. **Receiver** – Knows how to perform the action.
4. **Invoker** – Executes the command at a later time.

Analogy: Think of a **remote control** (Invoker). The buttons on the remote are **commands**, and the TV is the **receiver**. The user (client) just presses a button without worrying about the underlying implementation.

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, or NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

Code:

Step 1: Create interface	**public interface Order { void execute(); }**
Step 2: Create the Receiver class	**public class Stock { private String name = "ABC"; private int quantity = 10; public void buy() { System.out.println("Stock [Name: " + name + ", Quantity:" + quantity +"] bought"); } public void sell() { System.out.println("Stock [Name: " + name + ", Quantity:" + quantity +"] sold"); }}**
Step 3: Create Concrete Command	**public class BuyStock implements Order { private Stock abcStock;**

classesclass	**public BuyStock(Stock abcStock) { this.abcStock = abcStock; }** **@Override** **public void execute() { abcStock.buy(); }}** **public class SellStock implements Order {** **private Stock abcStock;** **public SellStock(Stock abcStock) {** **this.abcStock = abcStock; }** **@Override** **public void execute() { abcStock.sell(); }}**
Step 4: Create the Invoker class	**import java.util.ArrayList;** **import java.util.List;** **public class Broker {** **private List<Order> orderList = new ArrayList<Order>();** **public void takeOrder(Order order) {** **orderList.add(order); }** **public void placeOrders() {** **for (Order order : orderList) {** **order.execute(); } orderList.clear(); }}**
Step 5: Test the Command Pattern	**public class CommandPatternDemo {** **public static void main(String[] args) {** **Stock abcStock = new Stock();** **BuyStock buyStockOrder = new BuyStock(abcStock);** **SellStock sellStockOrder = new SellStock(abcStock);** **Broker broker = new Broker();** **broker.takeOrder(buyStockOrder);** **broker.takeOrder(sellStockOrder);** **broker.placeOrders(); }}**
Expected Output	**Stock [Name: ABC, Quantity:10] bought** **Stock [Name: ABC, Quantity:10] sold**

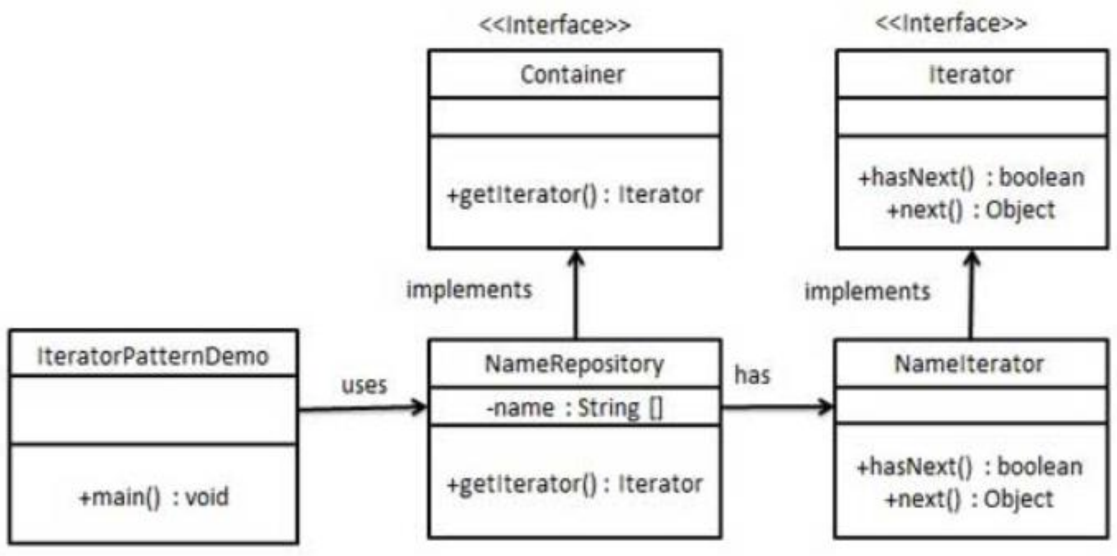
Result

The experiment successfully demonstrates the **Command Pattern** by encapsulating requests (BuyStock, SellStock) as command objects that are executed by the Broker (Invoker) without the client directly calling the Stock (Receiver).

Viva Questions

1. What is the Command Pattern and why is it used?
2. Which design pattern category does Command belong to?
3. Give a real-life example of the Command Pattern.
4. What is the difference between Command Pattern and Strategy Pattern?
5. How does the Command Pattern support undo/redo operations?

Experiment No. 11 Behavioral Design Patterns: • Iterator Pattern



Experiment Title

Implementation of Iterator Design Pattern in Java

Objective

- To understand the **Iterator Design Pattern**.
- To implement a program where the elements of a collection are accessed **sequentially** without exposing the underlying representation.
- To demonstrate how the Iterator provides a **standard way of traversing collections**.

Theory

- **Iterator Pattern** is a **behavioral design pattern**.

- It provides a **way to access elements of a collection sequentially** without exposing its internal structure.
- Commonly used in Java collections (List, Set, etc.).
- Example: When iterating through a **playlist of songs**, the user does not need to know how the playlist is stored internally.

In this experiment:

- **Iterator Interface** defines methods for traversing (hasNext(), next()).
- **Concrete Iterator** implements traversal logic.
- **Container Interface** defines a method to return an iterator.
- **Concrete Container** (collection) provides the iterator.

Software & Hardware Requirements

- **Software:** Java JDK 8 or later, IDE (Eclipse, IntelliJ, or NetBeans).
- **Hardware:** Standard computer system with at least 4GB RAM.

Code:

Step 1: Create interface	**public interface Iterator {** **boolean hasNext();** **Object next(); }**
Step 2: Create the Container interface	**public interface Container {** **Iterator getIterator(); }**
Step 3: Create the NameRepository class (Concrete Collection)	**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() {** **return index < names.length; }** **@Override** **public Object next() {** **if(this.hasNext()) {** **return names[index++]; } return null; } }}**
Step 5: Test	**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); } }}
Expected Output	**Name: Robert, Name: John, Name: Julie Name: Lora**

Result

The experiment successfully demonstrates the **Iterator Pattern** by providing a way to access elements of a collection (NameRepository) sequentially without exposing its internal representation.

Viva Questions

1. What is the Iterator Pattern and why is it used?
2. Which design pattern category does Iterator belong to?
3. How does Java's built-in Iterator interface relate to this pattern?
4. Can an Iterator be used to remove elements during traversal?
5. What is the difference between Iterator and Enumeration in Java?