K.S.Institute of Technology, Bengaluru-109

OO Design Patterns Laboratory
BCSL504

Prepared by,

Mrs Ammu Bhuvana D
Assistant Professor

Department of Computer Science and Design
K.S.Institute of Technology
Bangalore-109

K.S. INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & DESIGN

Vision of the Institute

To impart quality technical education with ethical values, employable skills and research to achieve excellence

Mission of the Institute

- To attract and retain highly qualified, experienced & committed faculty.
- To create relevant infrastructure.
- Network with industry & premier institutions to encourage emergence of new ideas by providing research & development facilities to strive for academic excellence.
- To inculcate the professional & ethical values among young students with employable skills & knowledge acquired to transform the society.

K.S. INSTITUTE OF TECHNOLOGY, BANGALORE - 560109

DEPARTMENT OF COMPUTER SCIENCE & DESIGN

Department Vision, Mission, PEOs, and PSOs

Vision:

To prepare skilled and distinct professionals in Computer Science & Design with good research skills backed by ethics for addressing societal needs.

Mission:

1. Instill a commitment to enhance innovative design skills on par with the technological advancements in the IT industry.
2. To make efforts to explore the research ideas in hardware and software design.
3. To groom the students with the design techniques to cater to the needs of society with professional ethics.

PEOs

1. Exhibit design skills and become an excellent employee in the interdisciplinary field.
2. Exhibit creative thinking and continue learning through higher studies and research.
3. Proficient in solving real-life problems related to innovation through teamwork and ethics.

PSOs

1. Ability to Utilize Concepts and Practices of computer science & design to develop solutions.
2. Ability to demonstrate design and development skills to solve problems in the broad area of programming concepts and real-world challenges.

OO Design Patterns Lab		Semester	**V**
Course Code	**BCGL504**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	100
Examination type (SEE)	Practical		

Course objectives:

- To introduce students to the fundamental principles and concepts of design patterns and their role inobject-oriented software development.
- To equip students with the skills to identify and apply the most appropriate design patterns to solve common software design problems.
- To develop the ability to analyze the advantages and disadvantages of different design patterns in real-world applications.
- To provide hands-on experience in implementing various design patterns using object-oriented programming languages.

Sl.NO	**Experiments (Implementation using Star UML)**
1	Design and implement ShapeFactory class that generates different types of Shape objects (Circle, Square, Rectangle) based on input parameters using Factory Design Pattern.
2	Design and Implement an AbstractFactory class to create families of related or dependent objects with respect to decathlon store without specifying their concreteclasses using Abstract Factory.
3	Design and implement a complex object like a House using a step-by-step Builderpattern, allowing different representations of the house (wooden, brick, etc.).
4	Design and Implement to Extend a Coffee object with dynamic features (e.g., milk,sugar, whipped cream) using Decorators.
5	Design and Implement a Logger class ensuring a single instance throughout theapplication
6	Design and implement an Adapter Pattern for a Music System.
7	Design and Implement an Observer pattern for a news agency to notify subscribersof updates.
8	Design and Implement a Façade pattern for home theatre system.
9	Design and Implement a Template Method for Document Processing (word, pdf,excel)
10	Design and Implement weather monitoring system that notifies multiple displaydevices whenever the weather conditions change that follows the Observer Design Pattern.
11	Design and Implement a Proxy pattern to control access to an object (e.g., a protectedresource or remote service).
12	Design and Implement a Mediator pattern to manage communication between a setof objects (e.g., chat room with multiple participants).

Course outcomes (Course Skill Set):

At the end of the course the student will be able to:

- Design the model for the given problem using UML concepts and notations.
- Develop the solution for the given real world problem using design patterns .
- Analyze the results and produce substantial written documentation.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

- Each experiment is to be evaluated for conduction with an observation sheet and recordwrite-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.
- Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.
- Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).
- Weightage to be given for neatness and submission of record/write-up on time.
- Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.
- In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.
- The suitable rubrics can be designed to evaluate each student's performance and learning ability.
- The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test isthe total CIE marks scored by the student.

Semester End Evaluation (SEE):

- SEE marks for the practical course are 50 Marks.
- SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.
- The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to beconducted between the schedule mentioned in the academic calendar of the University.
- All laboratory experiments are to be included for practical examination.
- (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners.

- **Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.**
- **Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.**

General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)
Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.
The minimum duration of SEE is 02 hours

Suggested Learning Resources:

- **Alan Shalloway, James R Trot, “Design Patterns Explained – A New Perspective on Object Oriented Design”, Pearson, 2nd Edition, 4th Impression 2010.**
- **Eric Freeman, Elisabeth Freeman, “Head First Design Patterns”, O’reilly Publications, October 2004, 1st Edition.**
- **Satzinger, Jackson, Burd, “Object Oriented Analysis and Design with Unified Process”, Thomson Learning, 1st Indian Reprint 2007.**
- **https://www.udemy.com/course/design-patterns-java/**
- **https://nptel.ac.in/courses/106105224**

CO-PO MAPPINGS					
Sl No	**Program List**	**CO**	**PO**	**RBT**	**Page No**
1	Design and implement Shape factory using Factory Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L 3	8-10
2	Design and implement Decathlon Store using Abstract Factory Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L 3	11-13
3	Design and implement House using Build Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L 3	14-17
4	Design and implement Coffee object using Decorator Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	18-21

5	Design and implement Logger class using Single Instance Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	22-23
6	Design and implement Music System using Adapter Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	24-26
7	Design and implement news agency using Observer Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	27-29
8	Design and implement Home Theater system using Façade Pattern	CO1,CO2,CO3,CO4,CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	30-33
9	Design and implement Document Processing using Template Pattern	CO1,CO2,CO3,CO4, CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	34-36

10	Design and implement Weather monitoring system using Observer design Pattern	CO1,CO2,CO3,CO4,CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	37-40
11	Design and implement remote database access using Proxy Pattern	CO1,CO2,CO3,CO4,CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	43-45
12	Design and implement chatroom system using Mediator Pattern	CO1,CO2,CO3,CO4,CO5	PO1 PO2 PO3 PO4 PO5 PO6 PO12	L1,L2,L3	46-46

C01: To learn the fundamentals principles and concepts of object oriented software development

CO2: To design the software model using standard UML notations in starUML

C03: To identify and apply design pattern for the given application or real world problems

C04: To provide hands on experience in implementing the various design patterns using object oriented language

C05: To learn the advantages and disadvantages of various design patterns

Cos	POS												
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PS O1	PS O2
CO1	2	2	2	2	2	2						2	2	2
CO2	2	2	2	2	2	2						2	2	2
CO3	2	2	2	2	2	2						2	2	2
CO4	2	2	2	2	2	2						2	2	2
CO5	2	2	2	2	2	2						2	2	2

Experiment 1

1. **Design and implement ShapeFactory class that generates different types of Shape objects (Circle, Square, Rectangle) based on input parameters using Factory Design Pattern.**

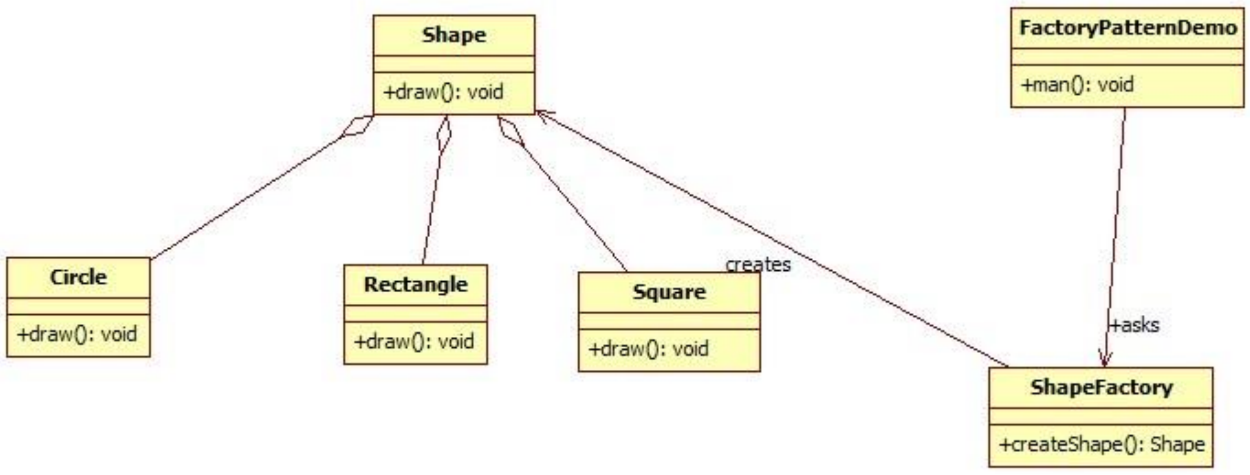
Solution:

```
//Circle.java
package client;

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

CircleFactory.java
package client;

public class CircleFactory implements ShapeFactory {
@Override
public Shape createShape() {
return new Circle();
}
}

//Client.java
package client;

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


ShapeFactory circleFactory = new CircleFactory();
Shape circle = circleFactory.createShape();
circle.draw();
ShapeFactory squareFactory = new SquareFactory();
Shape square = squareFactory.createShape();
square.draw();

ShapeFactory rectangleFactory = new RectangleFactory();
Shape rectangle = rectangleFactory.createShape();
rectangle.draw();
}
}

//Rectangle.java
package client;

public class Rectangle implements Shape {

@Override
public void draw() {
```

```
System.out.println("Drawing Rectangle");
}
}


//RectangleFactory.java
package client;

public class RectangleFactory implements ShapeFactory {
@Override
public Shape createShape() {
return new Rectangle();
}
}

//Shape.java

Package client;

public interface Shape {
void draw();
}

ShapeFactory.java
package client;

public interface ShapeFactory {

        Shape createShape();
}

//Square.java
package client;

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

//SquareFactory.java
package client;

public class SquareFactory implements ShapeFactory {
@Override
public Shape createShape() {
return new Square();


}
}
```

Output:

```
Drawing Circle
Drawing Square
Drawing Rectangle
```



Experiment 2

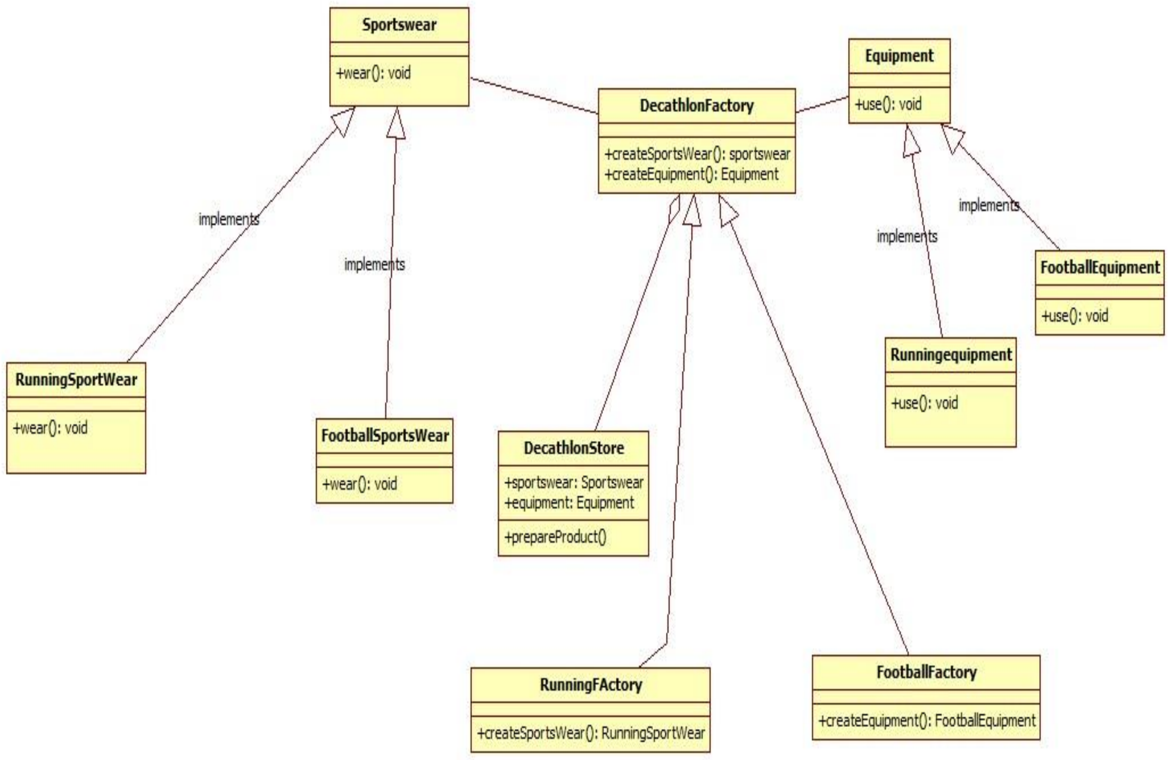
2. **Design and Implement an Abstract Factory class to create families of related or dependent objects with respect to decathlon store without specifying their concreteclasses using Abstract Factory**.

Solution:

```
// Abstract product: SportsWear
public interface SportsWear {
    void wear();
}

// Abstract product: Equipment
public interface Equipment {
    void use();
}

// Concrete product: RunningSportsWear
public class RunningSportsWear implements SportsWear {
    @Override
    public void wear() {
        System.out.println("Wearing running sportswear.");
    }
}

// Concrete product: FootballSportsWear
public class FootballSportsWear implements SportsWear {
    @Override
    public void wear() {
        System.out.println("Wearing football sportswear.");
    }
}

// Concrete product: RunningEquipment
public class RunningEquipment implements Equipment {
    @Override
    public void use() {
        System.out.println("Using running equipment.");
    }
}

// Concrete product: FootballEquipment
public class FootballEquipment implements Equipment {
    @Override
    public void use() {
        System.out.println("Using football equipment.");
    }

}
// Abstract factory
public interface DecathlonFactory {
    SportsWear createSportsWear();
    Equipment createEquipment();
}
// Concrete factory for Running products
public class RunningFactory implements DecathlonFactory {
    @Override
    public SportsWear createSportsWear() {
        return new RunningSportsWear();
    }

    @Override
    public Equipment createEquipment() {
        return new RunningEquipment();
    }
}

// Concrete factory for Football products
public class FootballFactory implements DecathlonFactory {
    @Override
    public SportsWear createSportsWear() {
        return new FootballSportsWear();
    }

    @Override
    public Equipment createEquipment() {
        return new FootballEquipment();
    }
}
public class DecathlonStore {
    private SportsWear sportsWear;
    private Equipment equipment;

    public DecathlonStore(DecathlonFactory factory) {
        sportsWear = factory.createSportsWear();
        equipment = factory.createEquipment();
    }

    public void prepareProducts() {
        sportsWear.wear();
        equipment.use();
    }

    public static void main(String[] args) {
        // Create products for running
        DecathlonFactory runningFactory = new RunningFactory();
        DecathlonStore runningStore = new DecathlonStore(runningFactory);
        System.out.println("Running Store:");
        runningStore.prepareProducts();

        // Create products for football
        DecathlonFactory footballFactory = new FootballFactory();
        DecathlonStore footballStore = new DecathlonStore(footballFactory);
        System.out.println("\nFootball Store:");
        footballStore.prepareProducts();
    }
}
```

Output:

```
Running Store:
Wearing running sportswear.
Using running equipment.

Football Store:
Wearing football sportswear.
Using football equipment.
```



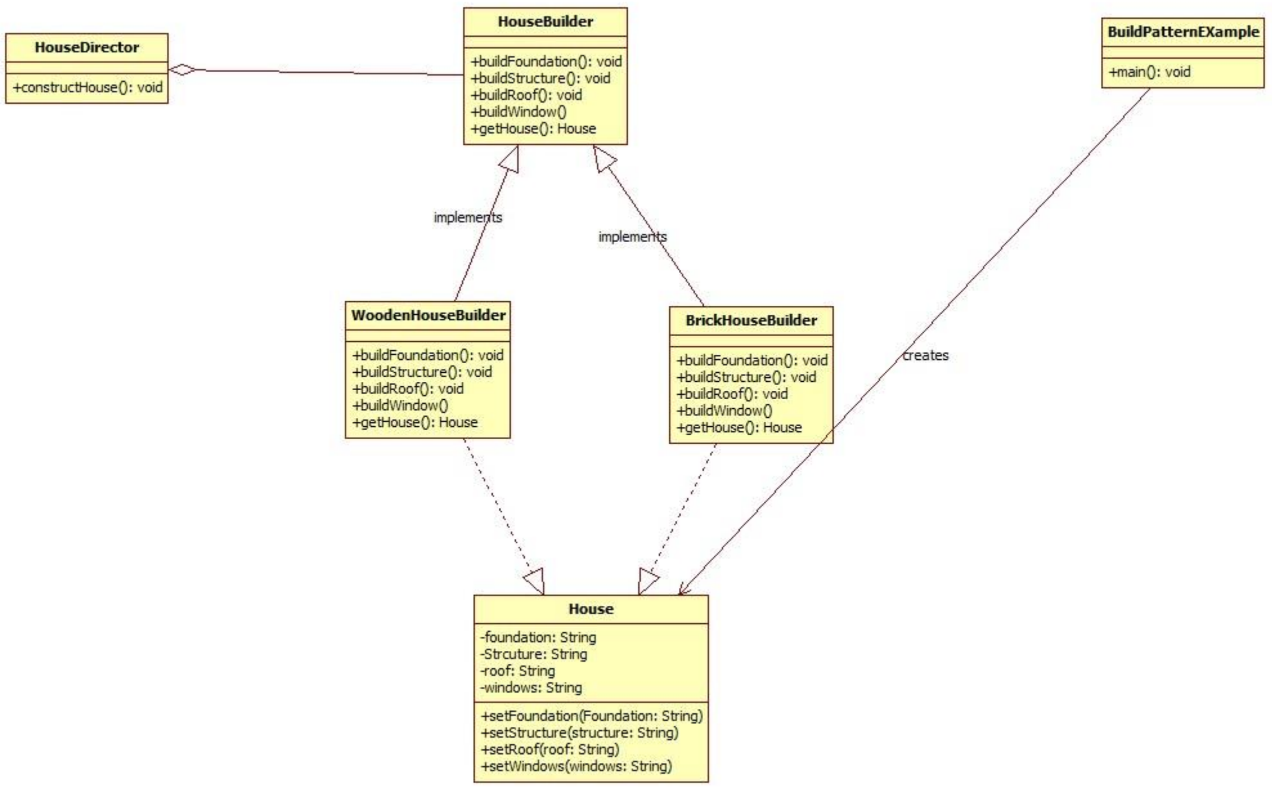
Experiment 3

3. **Design and implement a complex object like a House using a step-by-step Builder pattern, allowing different representations of the house (wooden, brick, etc.).**

```
public class House {
    private String foundation;
    private String structure;
    private String roof;
    private String windows;

    public void setFoundation(String foundation) {
        this.foundation = foundation;
    }

    public void setStructure(String structure) {
        this.structure = structure;
    }

    public void setRoof(String roof) {
        this.roof = roof;
    }

    public void setWindows(String windows) {
        this.windows = windows;
    }

    @Override
    public String toString() {
        return "House{" +
                "foundation='" + foundation + '\'' +
                ", structure='" + structure + '\'' +
                ", roof='" + roof + '\'' +
                ", windows='" + windows + '\'' +
                '}';
    }
}
public interface HouseBuilder {
    void buildFoundation();
    void buildStructure();
    void buildRoof();
    void buildWindows();
    House getHouse();
}
public class WoodenHouseBuilder implements HouseBuilder {
    private House house;

    public WoodenHouseBuilder() {

this.house = new House();
    }

    @Override
    public void buildFoundation() {
        house.setFoundation("Wooden Piles");
    }

    @Override
    public void buildStructure() {
        house.setStructure("Wooden Frame");
    }

    @Override
    public void buildRoof() {
        house.setRoof("Wooden Shingles");
    }

    @Override
    public void buildWindows() {
        house.setWindows("Wooden Windows");
    }

    @Override
    public House getHouse() {
        return this.house;
    }
}
public class BrickHouseBuilder implements HouseBuilder {
    private House house;

    public BrickHouseBuilder() {
        this.house = new House();
    }

    @Override
    public void buildFoundation() {
        house.setFoundation("Concrete Foundation with Steel Reinforcement");
    }

    @Override
    public void buildStructure() {
        house.setStructure("Brick Walls");
    }

    @Override
    public void buildRoof() {
        house.setRoof("Concrete Roof");
    }



@Override
    public void buildWindows() {
        house.setWindows("Aluminum Windows");
    }

    @Override
    public House getHouse() {
        return this.house;
    }
}
public class HouseDirector {
    private HouseBuilder houseBuilder;

    public HouseDirector(HouseBuilder houseBuilder) {
        this.houseBuilder = houseBuilder;
    }

    public void constructHouse() {
        houseBuilder.buildFoundation();
        houseBuilder.buildStructure();
        houseBuilder.buildRoof();
        houseBuilder.buildWindows();
    }

    public House getHouse() {
        return houseBuilder.getHouse();
    }
}
public class BuilderPatternExample {
    public static void main(String[] args) {
        // Construct a wooden house
        HouseBuilder woodenHouseBuilder = new WoodenHouseBuilder();
        HouseDirector director = new HouseDirector(woodenHouseBuilder);
        director.constructHouse();
        House woodenHouse = director.getHouse();
        System.out.println("Wooden House: " + woodenHouse);

        // Construct a brick house
        HouseBuilder brickHouseBuilder = new BrickHouseBuilder();
        director = new HouseDirector(brickHouseBuilder);
        director.constructHouse();
        House brickHouse = director.getHouse();
        System.out.println("Brick House: " + brickHouse);
    }
}
```

Output

```
Wooden House: House{foundation='Wooden Piles', structure='Wooden Frame', roof='Wooden
Shingles', windows='Wooden Windows'}


Brick House: House{foundation='Concrete Foundation with Steel Reinforcement', structure='Brick
Walls', roof='Concrete Roof', windows='Aluminum Windows'}
```


HouseDirector
+constructHouse(): void
HouseBuilder
+buildFoundation(): void
+buildStructure(): void
+buildRoof(): void
+buildWindow()
+getHouse(): House
BuildPatternEXample
+main(): void
implements
implements
creates
WoodenHouseBuilder
+buildFoundation(): void
+buildStructure(): void
+buildRoof(): void
+buildWindow()
+getHouse(): House
BrickHouseBuilder
+buildFoundation(): void
+buildStructure(): void
+buildRoof(): void
+buildWindow()
+getHouse(): House
House
-foundation: String
-Strcuture: String
-roof: String
-windows: String
+setFoundation(Foundation: String)
+setStructure(structure: String)
+setRoof(roof: String)
+setWindows(windows: String)

Experiment 4

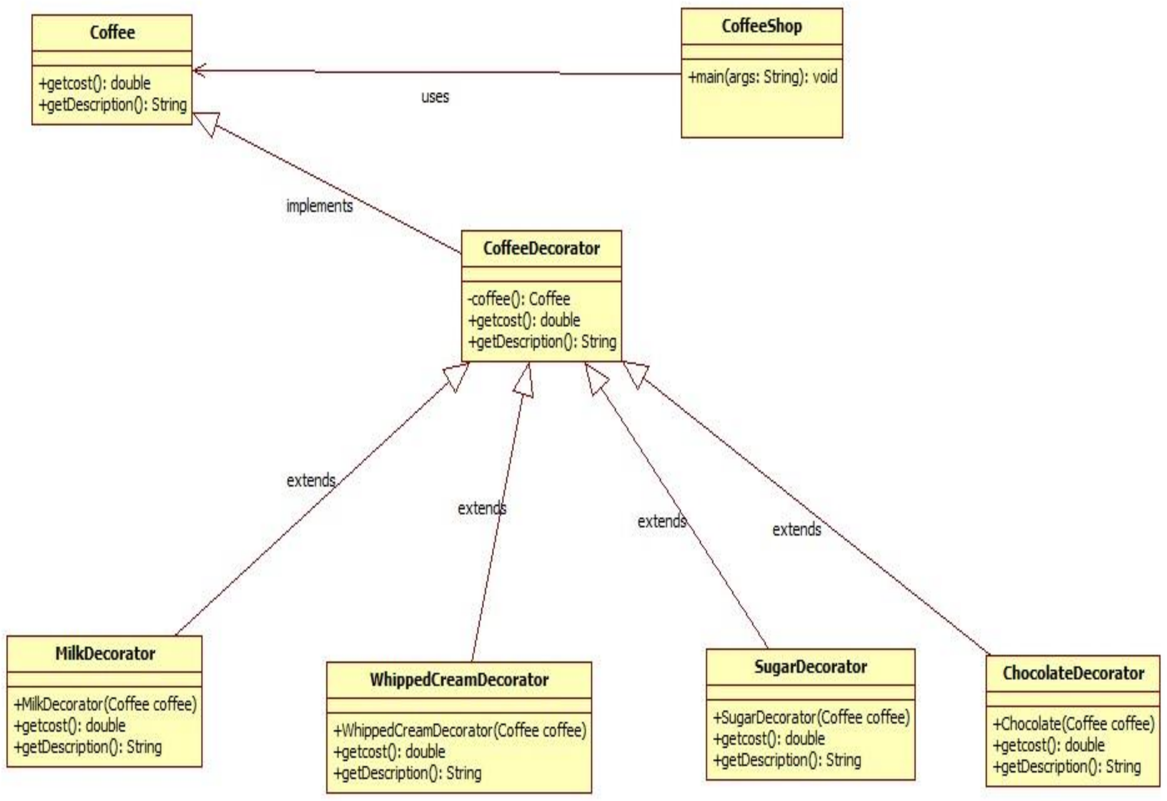
4. **Design and Implement to Extend a Coffee object with dynamic features (e.g., milk, sugar, whipped cream) using Decorators**

Solution:

```
// Coffee Interface (Component)
public interface Coffee {
   double getCost();      // Returns the cost of the coffee
   String getDescription(); // Returns the description of the coffee
}
// Concrete Component
public class SimpleCoffee implements Coffee {

   @Override
   public double getCost() {
      return 5.0; // base price of simple coffee
   }

   @Override
   public String getDescription() {
      return "Simple Coffee";
   }
}
// Abstract Decorator (implements Coffee interface)
public abstract class CoffeeDecorator implements Coffee {
   protected Coffee coffee; // The coffee object being decorated

   public CoffeeDecorator(Coffee coffee) {
      this.coffee = coffee;
   }

   @Override
   public double getCost() {
      return coffee.getCost(); // Forward request to decorated object
   }

   @Override
   public String getDescription() {
      return coffee.getDescription(); // Forward request to decorated object
   }
}
// Concrete Decorator (Milk)
public class MilkDecorator extends CoffeeDecorator {

   public MilkDecorator(Coffee coffee) {
      super(coffee);
   }


 @Override
   public double getCost() {
        return coffee.getCost() + 1.5; // Adding milk cost
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
}
// Concrete Decorator (Sugar)
public class SugarDecorator extends CoffeeDecorator {

    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.5; // Adding sugar cost
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }
}
// Concrete Decorator (Whipped Cream)
public class WhippedCreamDecorator extends CoffeeDecorator {

    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 2.0; // Adding whipped cream cost
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Whipped Cream";
    }
}
package CoffeeDecorator;

//Concrete Decorator (Sugar)
public class ChocolateDecorator extends CoffeeDecorator {

 public ChocolateDecorator(Coffee coffee) {


    super(coffee);
 }
 @Override
 public double getCost() {
    return coffee.getCost() + 0.5; // Adding sugar cost
 }

 @Override
 public String getDescription() {
    return coffee.getDescription() + ", Chocolate";
 }
}

public class CoffeeShop {
    public static void main(String[] args) {
        // Create a simple coffee
        Coffee myCoffee = new SimpleCoffee();
        System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());

        // Add milk to the coffee
        myCoffee = new MilkDecorator(myCoffee);
        System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());

        // Add sugar to the coffee
        myCoffee = new SugarDecorator(myCoffee);
        System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());

        // Add whipped cream to the coffee
        myCoffee = new WhippedCreamDecorator(myCoffee);
        System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());
// Add Chocolate Powder to the coffee
myCoffee = new ChocolateDecorator(myCoffee);
System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());
    }
}
```

Output:

```
Simple Coffee Cost: $5.0
Simple Coffee, Milk Cost: $6.5
Simple Coffee, Milk, Sugar Cost: $7.0
Simple Coffee, Milk, Sugar, Whipped Cream Cost: $9.0
Simple Coffee, Milk, Sugar, Whipped Cream, ChocolateCost:$9.5
```


Coffee
+getcost(): double
+getDescription(): String
CoffeeShop
+main(args: String): void
uses
implements
CoffeeDecorator
-coffee(): Coffee
+getcost(): double
+getDescription(): String
extends
extends
extends
extends
MilkDecorator
+MilkDecorator(Coffee coffee)
+getcost(): double
+getDescription(): String
WhippedCreamDecorator
+WhippedCreamDecorator(Coffee coffee)
+getcost(): double
+getDescription(): String
SugarDecorator
+SugarDecorator(Coffee coffee)
+getcost(): double
+getDescription(): String
ChocolateDecorator
+Chocolate(Coffee coffee)
+getcost(): double
+getDescription(): String

Experiment 5

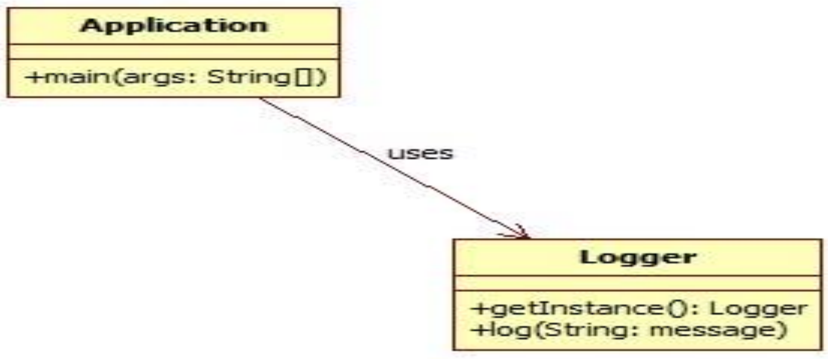
5. **Design and Implement a Logger class ensuring a single instance throughout the application**

Solution:

```
public class Logger {

    // Step 2: Create a static instance eagerly
    private static final Logger instance = new Logger();

    // Step 1: Make the constructor private to prevent instantiation
    private Logger() {
        // Optional: Add initialization code here
    }

    // Step 3: Provide a public method to access the single instance
    public static Logger getInstance() {
        return instance;
    }

    // Logger functionality
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
public class Application {
    public static void main(String[] args) {
        // Get the single Logger instance
        Logger logger = Logger.getInstance();

        // Use the logger to log messages
        logger.log("Application started.");
        logger.log("Application running smoothly.");
    }
}
```

Output:

```
Log: Application started.
Log: Application running smoothly.
```


Application
+main(args: String[])
uses
Logger
+getInstance(): Logger
+log(String: message)

Experiment 6

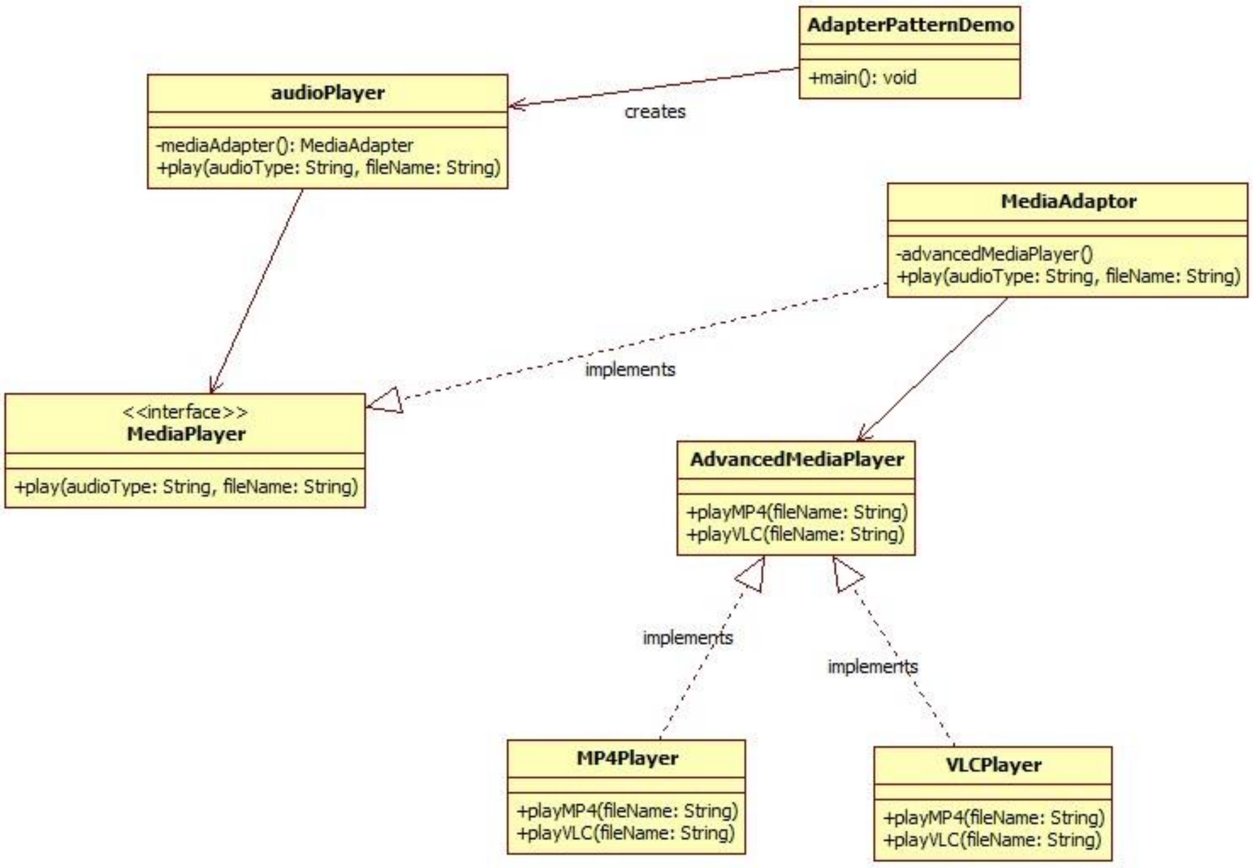
6. **Design and implement an Adapter Pattern for a Music System.**

Solution:

```
public interface MediaPlayer {
   void play(String audioType, String fileName);
}
public interface AdvancedMediaPlayer {
   void playMP4(String fileName);
   void playVLC(String fileName);
}
public class MP4Player implements AdvancedMediaPlayer {
   @Override
   public void playMP4(String fileName) {
      System.out.println("Playing MP4 file: " + fileName);
   }

   @Override
   public void playVLC(String fileName) {
      // Do nothing
   }
}

public class VLCPlayer implements AdvancedMediaPlayer {
   @Override
   public void playVLC(String fileName) {
      System.out.println("Playing VLC file: " + fileName);
   }

   @Override
   public void playMP4(String fileName) {
      // Do nothing
   }
}
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMediaPlayer;

   public MediaAdapter(String audioType) {
      if(audioType.equalsIgnoreCase("mp4")) {
         advancedMediaPlayer = new MP4Player();
      } else if(audioType.equalsIgnoreCase("vlc")) {
         advancedMediaPlayer = new VLCPlayer();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("mp4")) {
         advancedMediaPlayer.playMP4(fileName);
      } else if(audioType.equalsIgnoreCase("vlc")) {
         advancedMediaPlayer.playVLC(fileName);


      }
   }
}
public class AudioPlayer implements MediaPlayer {

    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {
        // Built-in support for mp3 files
        if(audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing MP3 file: " + fileName);
        }
        // MediaAdapter is used for other file formats
        else if(audioType.equalsIgnoreCase("mp4") || audioType.equalsIgnoreCase("vlc")) {
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        } else {
            System.out.println("Invalid media format: " + audioType + " not supported");
        }
    }
}
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "song.mp3");
        audioPlayer.play("mp4", "video.mp4");
        audioPlayer.play("vlc", "movie.vlc");
        audioPlayer.play("avi", "unsupported.avi");
    }
}
```

Output:

```
Playing MP3 file: song.mp3
Playing MP4 file: video.mp4
Playing VLC file: movie.vlc
Invalid media format: avi not supported
```


AdapterPatternDemo
+main(): void
creates
audioPlayer
-mediaAdapter(): MediaAdapter
+play(audioType: String, fileName: String)
MediaAdaptor
-advancedMediaPlayer()
+play(audioType: String, fileName: String)
implements
<<interface>>
MediaPlayer
+play(audioType: String, fileName: String)
AdvancedMediaPlayer
+playMP4(fileName: String)
+playVLC(fileName: String)
implements
implements
MP4Player
+playMP4(fileName: String)
+playVLC(fileName: String)
VLCPlayer
+playMP4(fileName: String)
+playVLC(fileName: String)

Experiment 7

7. **Design and Implement an Observer pattern for a news agency to notify subscribers of updates.**

Solution:

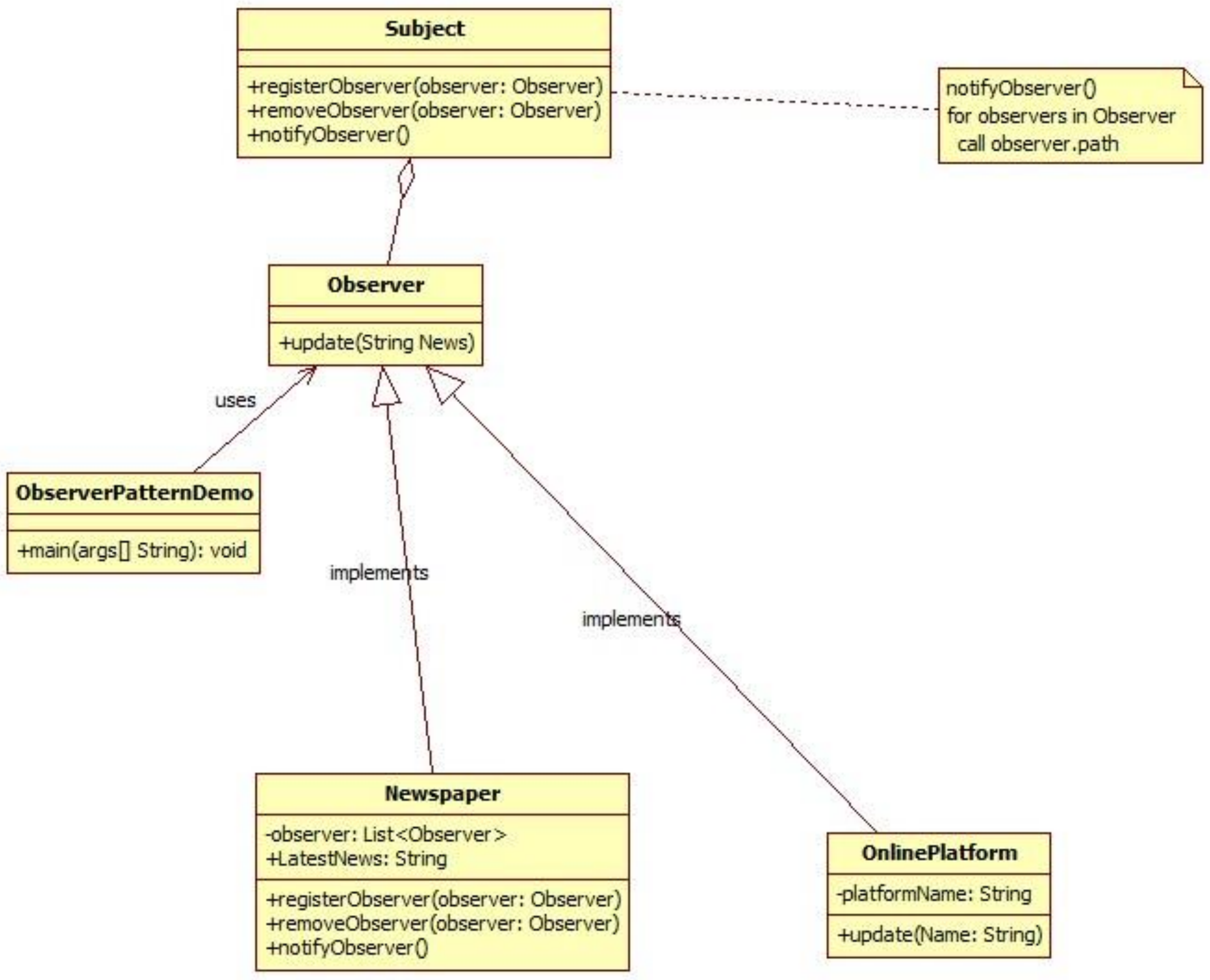
```
import java.util.List;

public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
public interface Observer {
    void update(String news);
}
import java.util.ArrayList;
import java.util.List;

public class NewsAgency implements Subject {
    private List<Observer> observers;
    private String latestNews;

    public NewsAgency() {
        this.observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(latestNews);
        }
    }

    public void setNews(String news) {
        this.latestNews = news;
        notifyObservers();
    }


}
```

```
public class Newspaper implements Observer {
    private String name;

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

    @Override
    public void update(String news) {
        System.out.println(name + " received news update: " + news);
    }
}
public class OnlinePlatform implements Observer {
    private String platformName;

    public OnlinePlatform(String platformName) {
        this.platformName = platformName;
    }

    @Override
    public void update(String news) {
        System.out.println(platformName + " updated with breaking news: " + news);
    }
}
public class ObserverPatternDemo {
    public static void main(String[] args) {
        // Create a news agency (Subject)
        NewsAgency newsAgency = new NewsAgency();

        // Create observers (Subscribers)
        Observer newspaper1 = new Newspaper("The Daily Times");
        Observer newspaper2 = new Newspaper("Morning News");
        Observer onlinePlatform = new OnlinePlatform("NewsNow");

        // Register the observers
        newsAgency.registerObserver(newspaper1);
        newsAgency.registerObserver(newspaper2);
        newsAgency.registerObserver(onlinePlatform);

        // Simulate news updates
        newsAgency.setNews("Breaking: New Java Version Released!");
        System.out.println();

        // Remove one observer and simulate another news update
        newsAgency.removeObserver(newspaper2);
        newsAgency.setNews("Weather Update: Heavy Rain Expected Tomorrow");
    }
}
```

Output:

```
The Daily Times received news update: Breaking: New Java Version Released!
Morning News received news update: Breaking: New Java Version Released!
```

```
NewsNow updated with breaking news: Breaking: New Java Version Released!

The Daily Times received news update: Weather Update: Heavy Rain Expected Tomorrow



NewsNow updated with breaking news: Weather Update: Heavy Rain Expected Tomorrow
```



Experiment 8

8. **Design and Implement a Façade pattern for home theatre system.**

Solution:

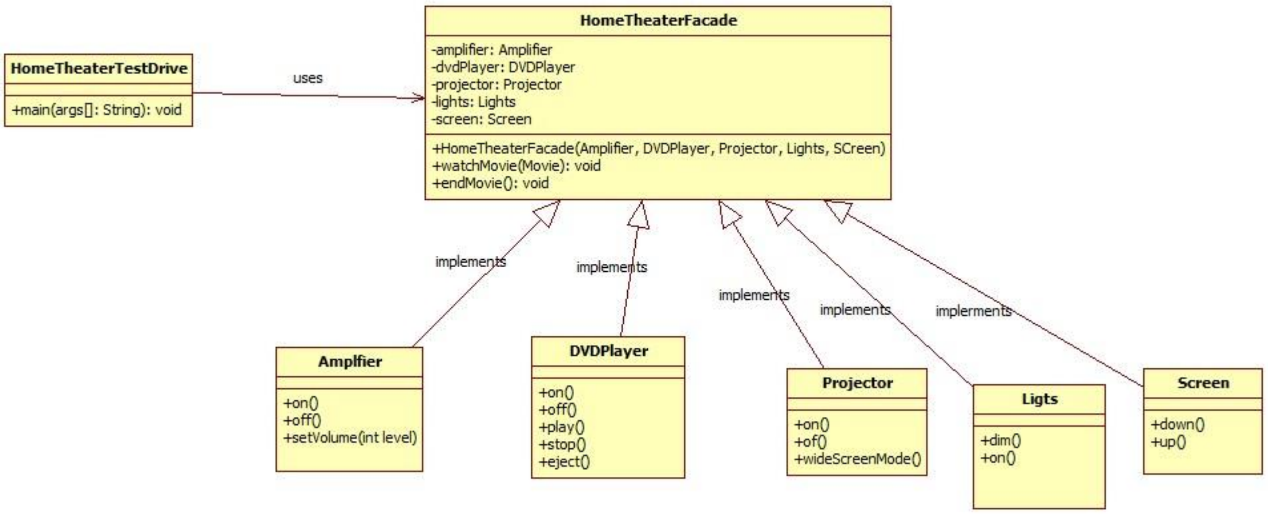
```
public class Amplifier {
    public void on() {
        System.out.println("Amplifier is on");
    }

    public void off() {
        System.out.println("Amplifier is off");
    }

    public void setVolume(int level) {
        System.out.println("Amplifier volume set to " + level);
    }
}
public class DVDPlayer {
    public void on() {
        System.out.println("DVD Player is on");
    }

    public void off() {
        System.out.println("DVD Player is off");
    }

    public void play(String movie) {
        System.out.println("Playing movie: " + movie);
    }

    public void stop() {
        System.out.println("Stopping the movie");
    }

    public void eject() {
        System.out.println("Ejecting the DVD");
    }
}
public class Projector {
    public void on() {
        System.out.println("Projector is on");
    }

    public void off() {
        System.out.println("Projector is off");
    }


  public void wideScreenMode() {
```

```
 System.out.println("Projector in widescreen mode (16x9 aspect ratio)");
    }
}
public class Lights {
    public void dim(int level) {
        System.out.println("Dimming the lights to " + level + "%");
    }

    public void on() {
        System.out.println("Lights are on");
    }
}
public class Screen {
    public void down() {
        System.out.println("Screen is going down");
    }

    public void up() {
        System.out.println("Screen is going up");
    }
}
public class HomeTheaterFacade {
    private Amplifier amplifier;
    private DVDPlayer dvdPlayer;
    private Projector projector;
    private Lights lights;
    private Screen screen;

    public HomeTheaterFacade(Amplifier amp, DVDPlayer dvd, Projector proj, Lights lights, Screen
screen) {
        this.amplifier = amp;
        this.dvdPlayer = dvd;
        this.projector = proj;
        this.lights = lights;
        this.screen = screen;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenMode();
        amplifier.on();
        amplifier.setVolume(5);
        dvdPlayer.on();
        dvdPlayer.play(movie);

    }

    public void endMovie() {
        System.out.println("Shutting movie theatre down...");
```

```
  lights.on();
      screen.up();
      projector.off();
      amplifier.off();
      dvdPlayer.stop();
      dvdPlayer.eject();
      dvdPlayer.off();
   }
}
public class HomeTheaterTestDrive {
   public static void main(String[] args) {
      // Create subsystem components
      Amplifier amp = new Amplifier();
      DVDPlayer dvd = new DVDPlayer();
      Projector projector = new Projector();
      Lights lights = new Lights();
      Screen screen = new Screen();

      // Create the Façade
      HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, dvd, projector, lights,
screen);

      // Watch a movie
      homeTheater.watchMovie("Inception");

      // End the movie
      homeTheater.endMovie();
   }
}
```

Output:

```
Amplifier volume set to 5
DVD Player is on
Playing movie: Inception
Shutting movie theatre down...
Lights are on
Screen is going up
Projector is off
Amplifier is off
Stopping the movie
Ejecting the DVD
DVD Player is off
```


HomeTheaterTestDrive
+main(args[]: String): void
uses
HomeTheaterFacade
-amplifier: Amplifier
-dvdPlayer: DVDPlayer
-projector: Projector
-lights: Lights
-screen: Screen
+HomeTheaterFacade(Amplifier, DVDPlayer, Projector, Lights, SCreen)
+watchMovie(Movie): void
+endMovie(): void
implements
implements
implements
implements
implerments
Amplfier
+on()
+off()
+setVolume(int level)
DVDPlayer
+on()
+off()
+play()
+stop()
+eject()
Projector
+on()
+of()
+wideScreenMode()
Ligts
+dim()
+on()
Screen
+down()
+up()

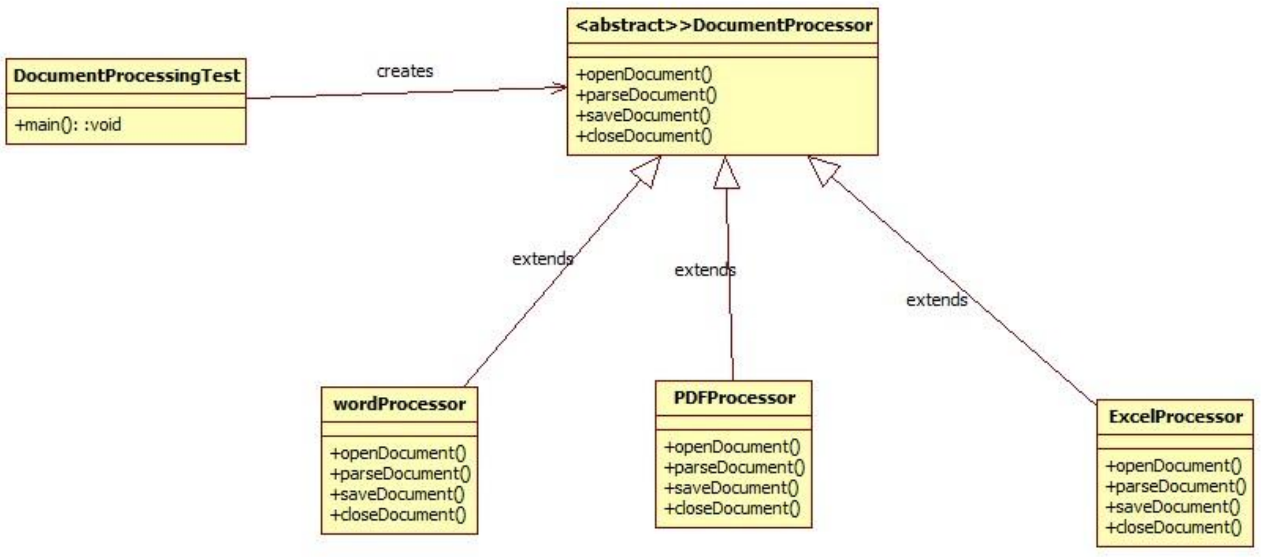
Experiment 9

9. **Design and Implement a Template Method for Document Processing (word, pdf, excel)**

Solution:

```
public abstract class DocumentProcessor {

    // Template Method
    public final void processDocument() {
        openDocument();
        parseDocument();
        saveDocument();
        closeDocument();
    }

    // Steps to be implemented by subclasses
    protected abstract void openDocument();
    protected abstract void parseDocument();
    protected abstract void saveDocument();
    protected abstract void closeDocument();
}
public class WordProcessor extends DocumentProcessor {

    @Override
    protected void openDocument() {
        System.out.println("Opening Word document...");
    }

    @Override
    protected void parseDocument() {
        System.out.println("Parsing Word document content...");
    }

    @Override
    protected void saveDocument() {
        System.out.println("Saving Word document...");
    }

    @Override
    protected void closeDocument() {
        System.out.println("Closing Word document.");
    }
}
public class PDFProcessor extends DocumentProcessor {

    @Override
    protected void openDocument() {
        System.out.println("Opening PDF document...");
    }

    @Override

    protected void parseDocument() {
        System.out.println("Parsing PDF document content...");
    }
    @Override
    protected void saveDocument() {
        System.out.println("Saving PDF document...");
    }

    @Override
    protected void closeDocument() {
        System.out.println("Closing PDF document.");
    }
}
public class ExcelProcessor extends DocumentProcessor {

    @Override
    protected void openDocument() {
        System.out.println("Opening Excel spreadsheet...");
    }

    @Override
    protected void parseDocument() {
        System.out.println("Parsing Excel spreadsheet content...");
    }

    @Override
    protected void saveDocument() {
        System.out.println("Saving Excel spreadsheet...");
    }

    @Override
    protected void closeDocument() {
        System.out.println("Closing Excel spreadsheet.");
    }
}
public class DocumentProcessingTest {
    public static void main(String[] args) {
        // Process a Word document
        DocumentProcessor wordProcessor = new WordProcessor();
        System.out.println("Processing Word Document:");
        wordProcessor.processDocument();

        System.out.println();

        // Process a PDF document
        DocumentProcessor pdfProcessor = new PDFProcessor();
        System.out.println("Processing PDF Document:");
        pdfProcessor.processDocument();

        System.out.println();


       // Process an Excel document
        DocumentProcessor excelProcessor = new ExcelProcessor();
        System.out.println("Processing Excel Document:");
        excelProcessor.processDocument();
    }
}
```

Output:
```
Opening PDF document...
Parsing PDF document content...
Saving PDF document...
Closing PDF document.

Processing Excel Document:
Opening Excel spreadsheet...
Parsing Excel spreadsheet content...
Saving Excel spreadsheet...
Closing Excel spreadsheet.
```



Experiment 10

10. **Design and Implement weather monitoring system that notifies multiple display devices whenever the weather conditions change that follows the Observer Design Pattern.**

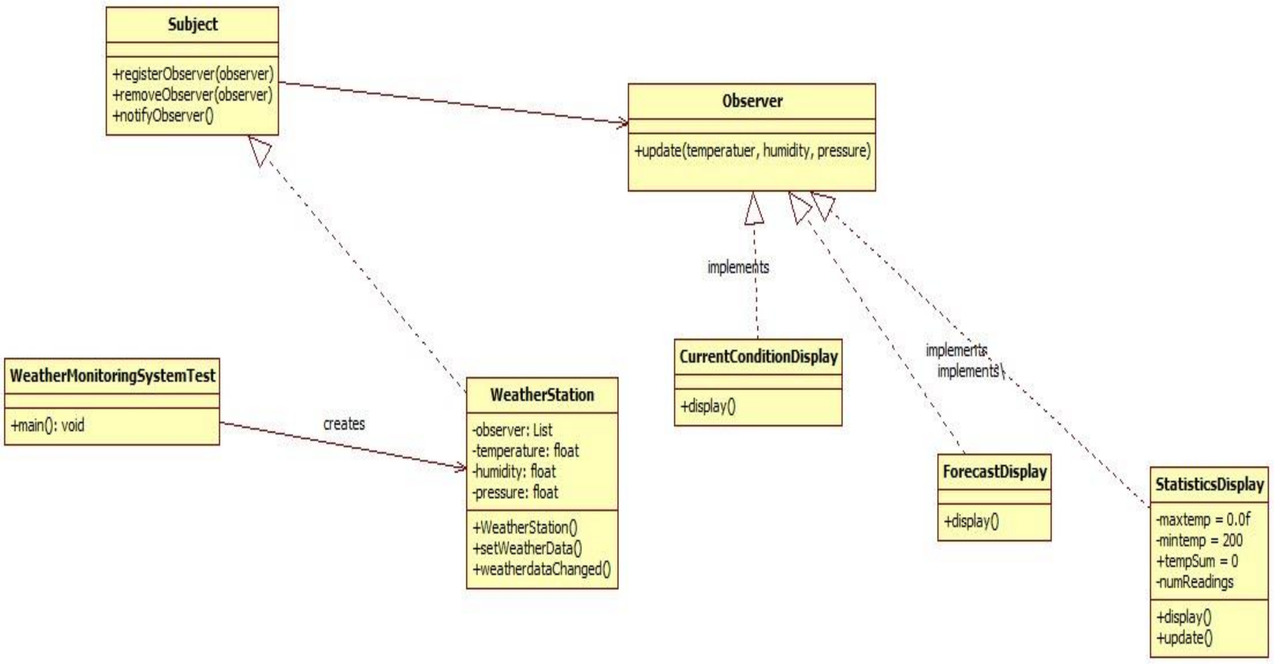
Solution:

```
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
public interface Observer {
    void update(float temperature, float humidity, float pressure);
}
import java.util.ArrayList;
import java.util.List;

public class WeatherStation implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherStation() {
        this.observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void setWeatherData(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        weatherDataChanged();


    }
    private void weatherDataChanged() {
        notifyObservers();
    }
}
public class CurrentConditionsDisplay implements Observer {
    private float temperature;
    private float humidity;

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature + "C degrees and " + humidity + "%
humidity");
    }
}
public class StatisticsDisplay implements Observer {
    private float maxTemp = 0.0f;
    private float minTemp = 200;
    private float tempSum = 0.0f;
    private int numReadings;

    @Override
    public void update(float temperature, float humidity, float pressure) {
        tempSum += temperature;
        numReadings++;

        if (temperature > maxTemp) {
            maxTemp = temperature;
        }

        if (temperature < minTemp) {
            minTemp = temperature;
        }

        display();
    }

    public void display() {
        System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings) + "/" +
maxTemp + "/" + minTemp);
    }
}

public class ForecastDisplay implements Observer {

  private float lastPressure;
    private float currentPressure = 29.92f;
    @Override
    public void update(float temperature, float humidity, float pressure) {
        lastPressure = currentPressure;
        currentPressure = pressure;
        display();
    }

    public void display() {
        System.out.print("Forecast: ");
        if (currentPressure > lastPressure) {
            System.out.println("Improving weather on the way!");
        } else if (currentPressure == lastPressure) {
            System.out.println("More of the same");
        } else if (currentPressure < lastPressure) {
            System.out.println("Watch out for cooler, rainy weather");
        }
    }
}
public class WeatherMonitoringSystemTest {
    public static void main(String[] args) {
        // Create WeatherStation (Subject)
        WeatherStation weatherStation = new WeatherStation();

        // Create display devices (Observers)
        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay();
        ForecastDisplay forecastDisplay = new ForecastDisplay();

        // Register displays as observers
        weatherStation.registerObserver(currentDisplay);
        weatherStation.registerObserver(statisticsDisplay);
        weatherStation.registerObserver(forecastDisplay);

        // Simulate new weather measurements
        weatherStation.setWeatherData(25.5f, 65.0f, 30.4f);
        System.out.println();

        weatherStation.setWeatherData(27.0f, 70.0f, 29.2f);
        System.out.println();

        // Remove one observer and simulate another weather update
        weatherStation.removeObserver(forecastDisplay);
        weatherStation.setWeatherData(22.0f, 60.0f, 28.4f);
    }
}
```

Output:

```
Current conditions: 25.5C degrees and 65.0% humidity
Avg/Max/Min temperature = 25.5/25.5/25.5
Forecast: Improving weather on the way!
Current conditions: 27.0C degrees and 70.0% humidity
Avg/Max/Min temperature = 26.25/27.0/25.5
```

```
Forecast: Watch out for cooler, rainy weather

Current conditions: 22.0C degrees and 60.0% humidity
Avg/Max/Min temperature = 24.833334/27.0/22.0
```



Experiment 11

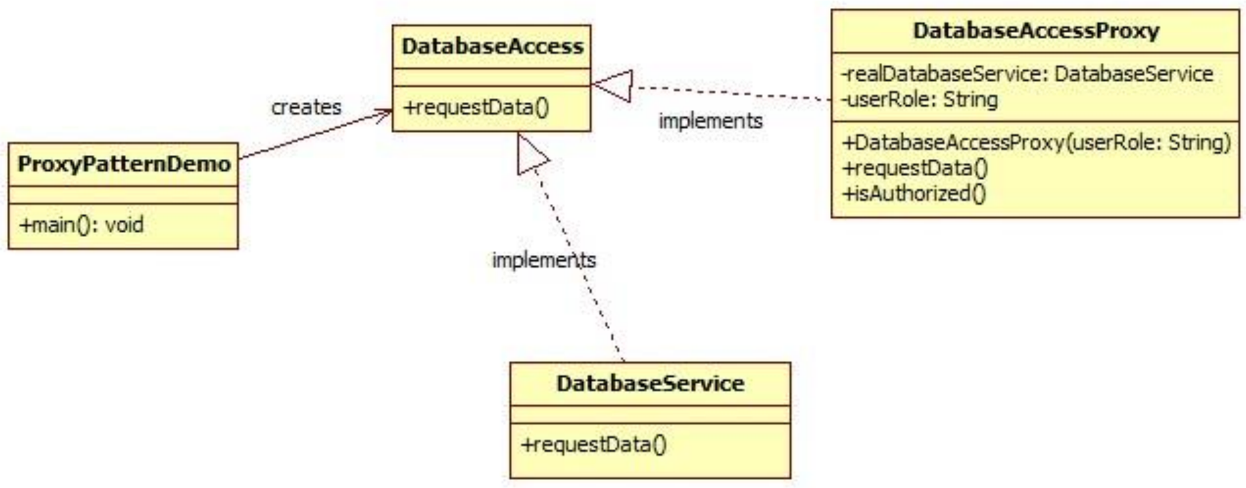
11. **Design and Implement a Proxy pattern to control access to an object (e.g., a protected resource or remote service).**

Solution:

```
public interface DatabaseAccess {
   void requestData();
}
public class DatabaseService implements DatabaseAccess {

   @Override
   public void requestData() {
      System.out.println("Fetching data from the database...");
   }
}
public class DatabaseAccessProxy implements DatabaseAccess {
   private DatabaseService realDatabaseService;
   private String userRole;

   public DatabaseAccessProxy(String userRole) {
      this.realDatabaseService = new DatabaseService();
      this.userRole = userRole;
   }

   @Override
   public void requestData() {
      if (isAuthorized()) {
         realDatabaseService.requestData();
      } else {
         System.out.println("Error: Access Denied. You do not have permission to access this resource.");
      }
   }

   private boolean isAuthorized() {
      return "ADMIN".equals(userRole);
   }
}
public class ProxyPatternDemo {
   public static void main(String[] args) {
      // User with ADMIN role
      DatabaseAccess adminAccess = new DatabaseAccessProxy("ADMIN");
      System.out.println("Admin trying to access the database:");
      adminAccess.requestData();

      System.out.println();

      // User with GUEST role (unauthorized)
      DatabaseAccess guestAccess = new DatabaseAccessProxy("GUEST");
      System.out.println("Guest trying to access the database:");
      guestAccess.requestData();
   }

}
```

Output

```
Admin trying to access the database:
Fetching data from the database...

Guest trying to access the database:
Error: Access Denied. You do not have permission to access this resource.
```



Experiment 12

12. **Design and Implement a Mediator pattern to manage communication between a setof objects (e.g., chat room with multiple participants).**

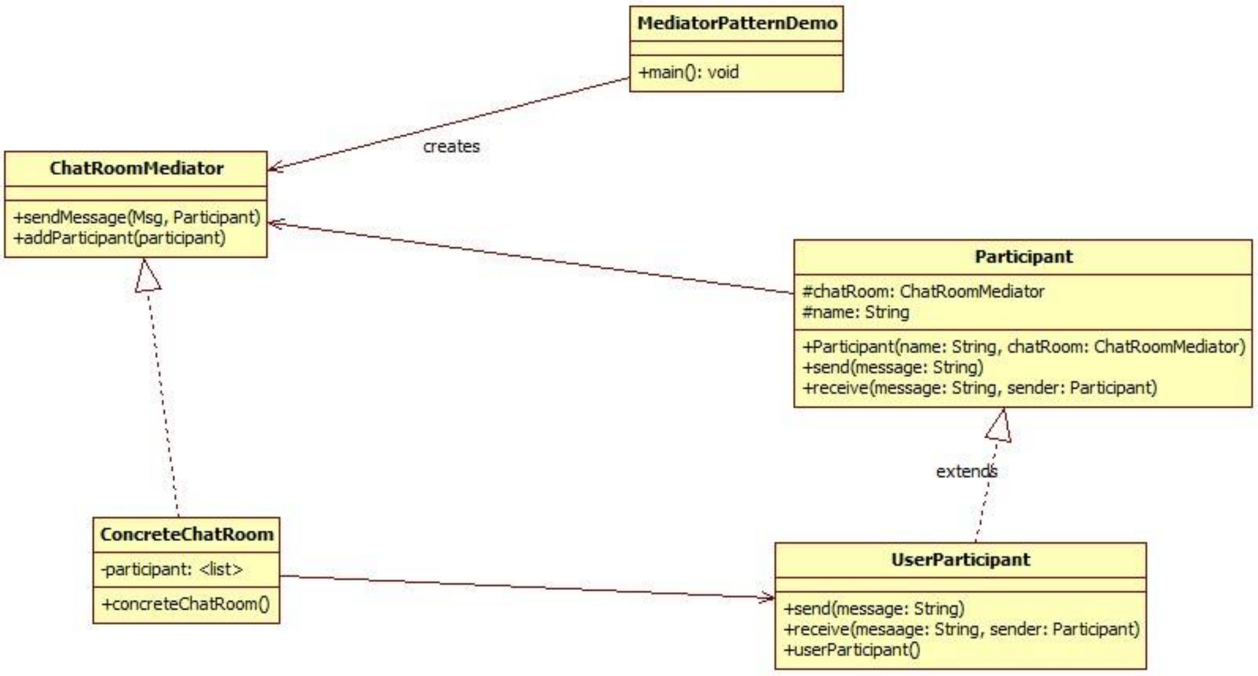
Solution:

```
public interface ChatRoomMediator {
    void sendMessage(String message, Participant participant);
    void addParticipant(Participant participant);
}
import java.util.ArrayList;
import java.util.List;

public class ConcreteChatRoom implements ChatRoomMediator {
    private List<Participant> participants;

    public ConcreteChatRoom() {
        this.participants = new ArrayList<>();
    }

    @Override
    public void sendMessage(String message, Participant sender) {
        for (Participant participant : participants) {
            // The sender should not receive the message it sent
            if (participant != sender) {
                participant.receive(message, sender);
            }
        }
    }

    @Override
    public void addParticipant(Participant participant) {
        participants.add(participant);
    }
}
public abstract class Participant {
    protected ChatRoomMediator chatRoom;
    protected String name;

    public Participant(String name, ChatRoomMediator chatRoom) {
        this.name = name;
        this.chatRoom = chatRoom;
    }

    public abstract void send(String message);
    public abstract void receive(String message, Participant sender);
}
public class UserParticipant extends Participant {

    public UserParticipant(String name, ChatRoomMediator chatRoom) {
        super(name, chatRoom);
    }


    @Override

                                public void send(String message) {
```

```
 System.out.println(this.name + " sends: " + message);
        chatRoom.sendMessage(message, this);
    }

    @Override
    public void receive(String message, Participant sender) {
        System.out.println(this.name + " receives a message from " + sender.name + ": " + message);
    }
}
public class MediatorPatternDemo {
    public static void main(String[] args) {
        // Create the mediator (Chat Room)
        ChatRoomMediator chatRoom = new ConcreteChatRoom();

        // Create participants
        Participant user1 = new UserParticipant("Alice", chatRoom);
        Participant user2 = new UserParticipant("Bob", chatRoom);
        Participant user3 = new UserParticipant("Charlie", chatRoom);

        // Add participants to the chat room
        chatRoom.addParticipant(user1);
        chatRoom.addParticipant(user2);
        chatRoom.addParticipant(user3);

        // Users send messages
        user1.send("Hello, everyone!");
        user2.send("Hey, Alice!");
        user3.send("Hi, Alice and Bob!");
    }
}
```

Output:

```
Alice sends: Hello, everyone!
Bob receives a message from Alice: Hello, everyone!
Charlie receives a message from Alice: Hello, everyone!
Bob sends: Hey, Alice!
Alice receives a message from Bob: Hey, Alice!
Charlie receives a message from Bob: Hey, Alice!
Charlie sends: Hi, Alice and Bob!
Alice receives a message from Charlie: Hi, Alice and Bob!
Bob receives a message from Charlie: Hi, Alice and Bob!
```


MediatorPatternDemo
+main(): void
creates
ChatRoomMediator
+sendMessage(Msg, Participant)
+addParticipant(participant)
Participant
#chatRoom: ChatRoomMediator
#name: String
+Participant(name: String, chatRoom: ChatRoomMediator)
+send(message: String)
+receive(message: String, sender: Participant)
extends
ConcreteChatRoom
-participant: <list>
+concreteChatRoom()
UserParticipant
+send(message: String)
+receive(mesaage: String, sender: Participant)
+userParticipant()

Viva Questions

1. **What is object oriented modeling and Design?**
2. **What is Pattern?**
3. **Mention the classification of pattern?**
4. **What is Creational pattern?**
5. **What is Structural patterns?**
6. **What is Behavioral pattern?**
7. **What is Factory Method?**
8. **Explain adapter pattern.**
9. **Explain builder pattern.**
10. **Explain singleton pattern.**
11. **Explain observer pattern?**
12. **What is StarUML?**
13. **Define class.**
14. **Define packages.**
15. **What is interface?**
16. **What are the characteristics of java?**