

ATME COLLEGE OF ENGINEERING

13th KM Stone, Mysuru, Kanakapura-Bangalore Road - 570 028



DEPARTMENT OF COMPUTER SCIENCE & DESIGN

**LABORATORY MANUAL OF
OBJECT ORIENTED DESIGN PATTERN LABORATORY
ACADEMIC YEAR 2026**

SUBJECT CODE: BCGL504

[As per Choice Based Credit System (CBCS) scheme]

SEMESTER-V

Prepared by

Verified by

Approved by

Mr. MAHADEVASWAMY

Ms. DARSHINI Y

Dr. NASREEN FATHIMA

PROGRAMMER

FACULTY CO-ORDINATOR

HOD, CS&DESIGN

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in Local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

OO Design Patterns Lab		Semester	5
Course Code	BCGL504	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	2
Examination type (SEE)	Practical		
Course objectives: • To introduce students to the fundamental principles and concepts of design patterns and their role in object-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		
1	Design and implement Shape Factory class that generates different types of Shape objects (Circle, Square, Rectangle) based on input parameters using Factory Design Pattern.		
2	Design and Implement an Abstract Factory class to create families of related or dependent objects with respect to decathlon store without specifying their concrete classes using Abstract Factory.		
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.).		
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 the application		
6	Design and implement an Adapter Pattern for a Music System.		
7	Design and Implement an Observer pattern for a news agency to notify subscribers of 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 display devices 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 protected resource or remote service).		
12	Design and Implement a Mediator pattern to manage communication between a set of 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:

1. Design the model for the given problem using UML concepts and notations.
2. Develop the solution for the given real world problem using design patterns .
3. 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 record write-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 is the 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 be conducted 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.

- 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 be conducted 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

CONTENTS

Sl. No.	Particulars	Page No
1	Introduction	1
2	Design and implement Shape Factory class that generates different types of Shape objects (Circle, Square, Rectangle) based on input parameters using Factory Design Pattern	7
3	Design and Implement an Abstract Factory class to create families of related or dependent objects with respect to decathlon store without specifying their concrete classes using Abstract Factory.	9
4	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.).	11
5	Design and Implement to Extend a Coffee object with dynamic features (e.g., milk, sugar, whipped cream) using Decorators.	15
6	Design and Implement a Logger class ensuring a single instance throughout the application	17
7	Design and implement an Adapter Pattern for a Music System.	20
8	Design and Implement an Observer pattern for a news agency to notify subscribers of updates.	22
9	Design and Implement a Façade pattern for home theatre system.	25
10	Design and Implement a Template Method for Document Processing (word, pdf, excel)	27
11	Design and Implement weather monitoring system that notifies multiple display devices whenever the weather conditions change that follows the Observer Design Pattern.	29
12	Design and Implement a Proxy pattern to control access to an object (e.g., a protected resource or remote service).	32
13	Design and Implement a Mediator pattern to manage communication between a set of objects (e.g., chat room with multiple participants).	36
14	Viva Questions and Answers	38

Experiment 1

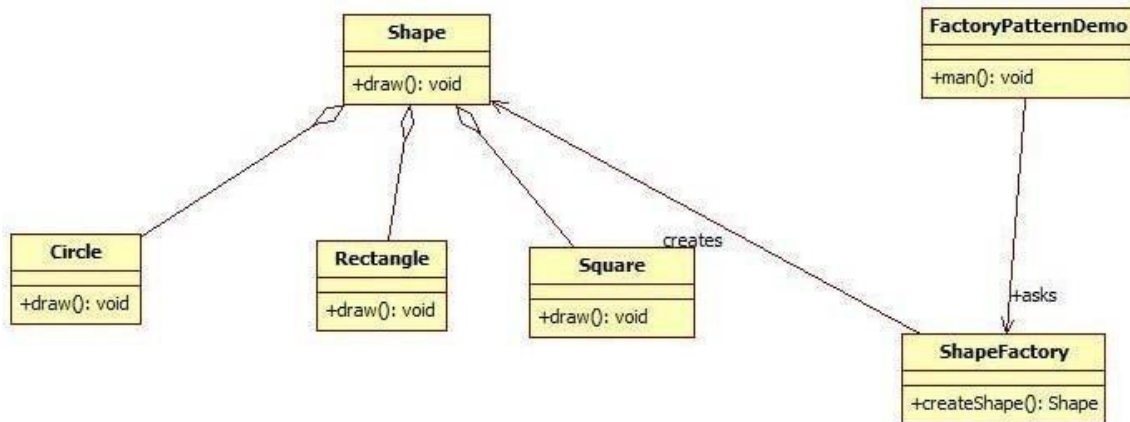
1. **Design and implement Shape Factory class that generates different types of Shape objects based on input parameters using Factory Design Pattern.**

Solution:

```
interface Shape {  
    void draw();  
}  
  
class Circle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Drawing Circle");  
    }  
}  
  
interface ShapeFactory {  
    Shape createShape();  
}  
  
class CircleFactory implements ShapeFactory {  
    @Override  
    public Shape createShape() {  
        return new Circle();  
    }  
}  
  
class Client {  
    public static void main(String[] args) {  
        ShapeFactory circleFactory = new CircleFactory();  
        Shape circle = circleFactory.createShape();  
        circle.draw(); // Only drawing circle  
    }  
}
```

Output:

Drawing Circle



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 concrete classes using Abstract Factory.**

Solution:

```
interface SportsWear {  
    void wear();  
}  
  
interface Equipment {  
    void use();  
}  
  
class RunningSportsWear implements SportsWear {  
    @Override  
    public void wear() {  
        System.out.println("Wearing running sportswear.");  
    }  
}  
  
class RunningEquipment implements Equipment {  
    @Override  
    public void use() {  
        System.out.println("Using running equipment.");  
    }  
}  
  
interface DecathlonFactory {  
    SportsWear createSportsWear();  
    Equipment createEquipment();  
}
```

```
class RunningFactory implements DecathlonFactory {

    @Override

    public SportsWear createSportsWear() {

        return new RunningSportsWear();

    }

    @Override

    public Equipment createEquipment() {

        return new RunningEquipment();

    }

}

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) {

        DecathlonFactory runningFactory = new RunningFactory();

        DecathlonStore runningStore = new DecathlonStore(runningFactory);

        System.out.println("Running Store:");

        runningStore.prepareProducts();

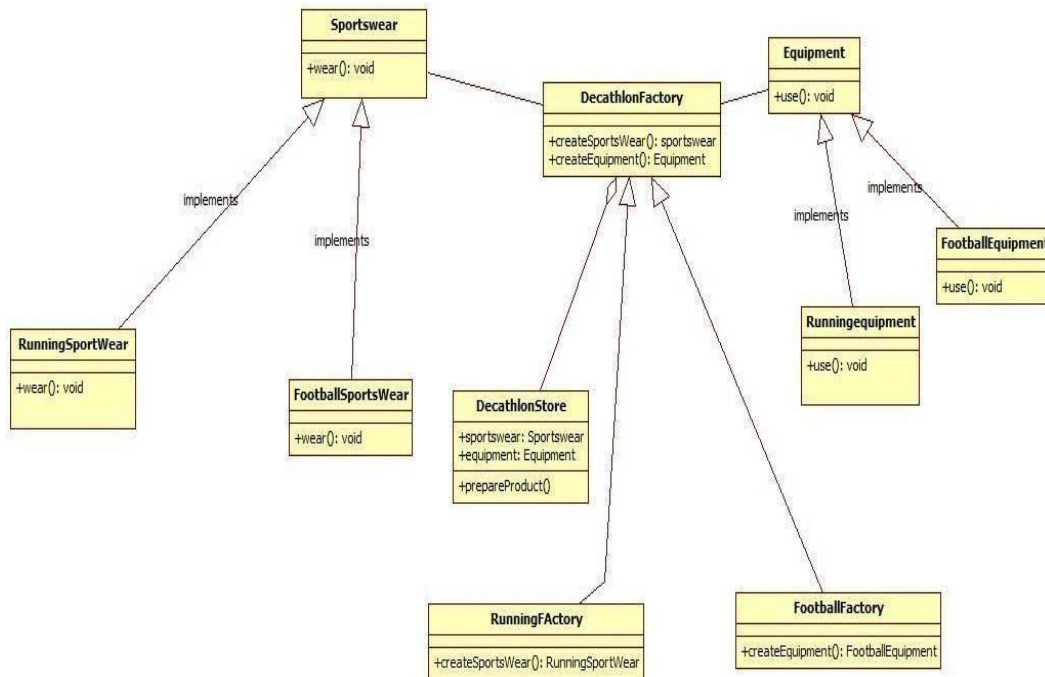
    }

}
```

```
}
```

Output:

Running Store:
Wearing running sportswear.
Using running 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.).**

```
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 + "\" + '}'";
    }
}

interface HouseBuilder { void buildFoundation(); void buildStructure(); void buildRoof();
void buildWindows(); House getHouse();
}

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;
}
}

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();
    }
}
```

```

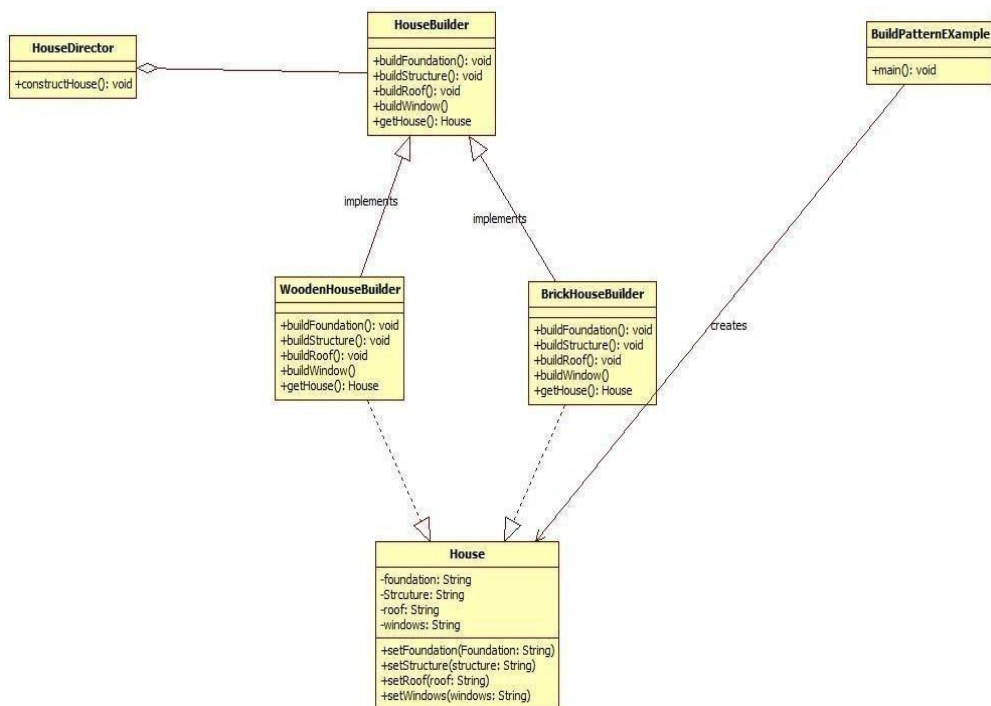
class BuilderPatternExample {
public static void main(String[] args) {

    HouseBuilder woodenHouseBuilder = new WoodenHouseBuilder();
    HouseDirector director = new HouseDirector(woodenHouseBuilder);
    director.constructHouse();
    House woodenHouse = director.getHouse();
    System.out.println("Wooden House: " + woodenHouse);
}
}

```

Output

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



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

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

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";

    }
}

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);

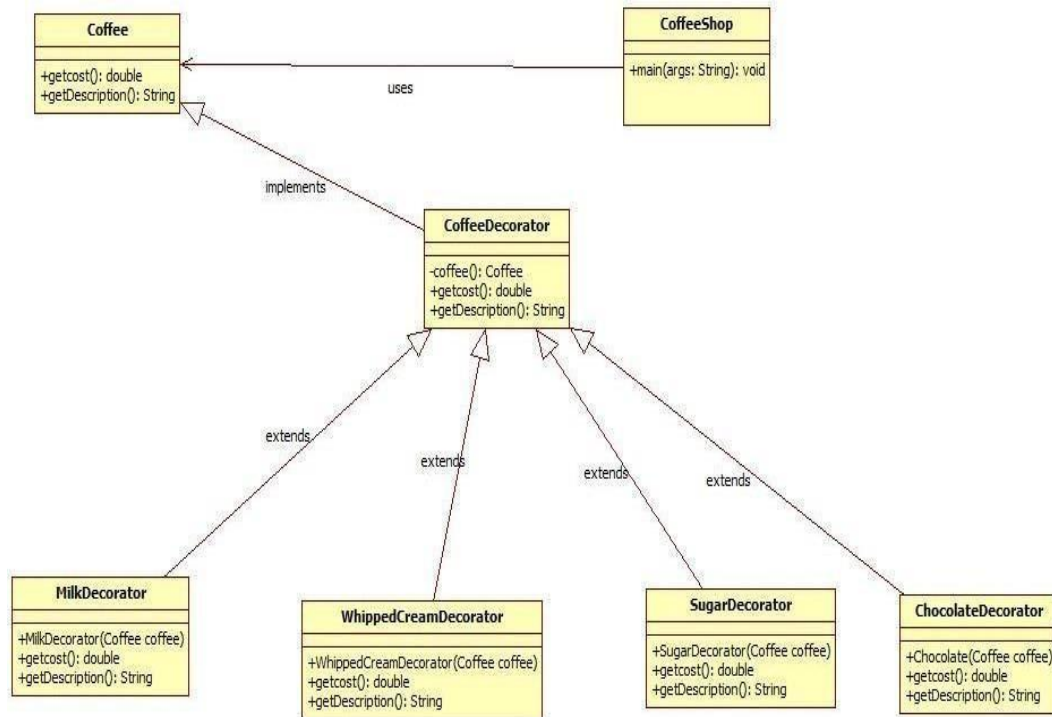
        System.out.println(myCoffee.getDescription() + " Cost: $" + myCoffee.getCost());

    }
}
```

Output:

Simple Coffee Cost: \$5.0

Simple Coffee, Milk Cost: \$6.5



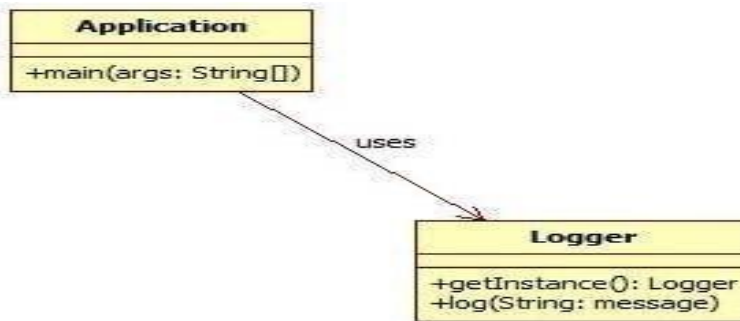
Experiment 5**5. Design and Implement a Logger class ensuring a single instance throughout the application****Solution:**

```
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);  
    }  
}  
class LoggerProgram {  
  
    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.

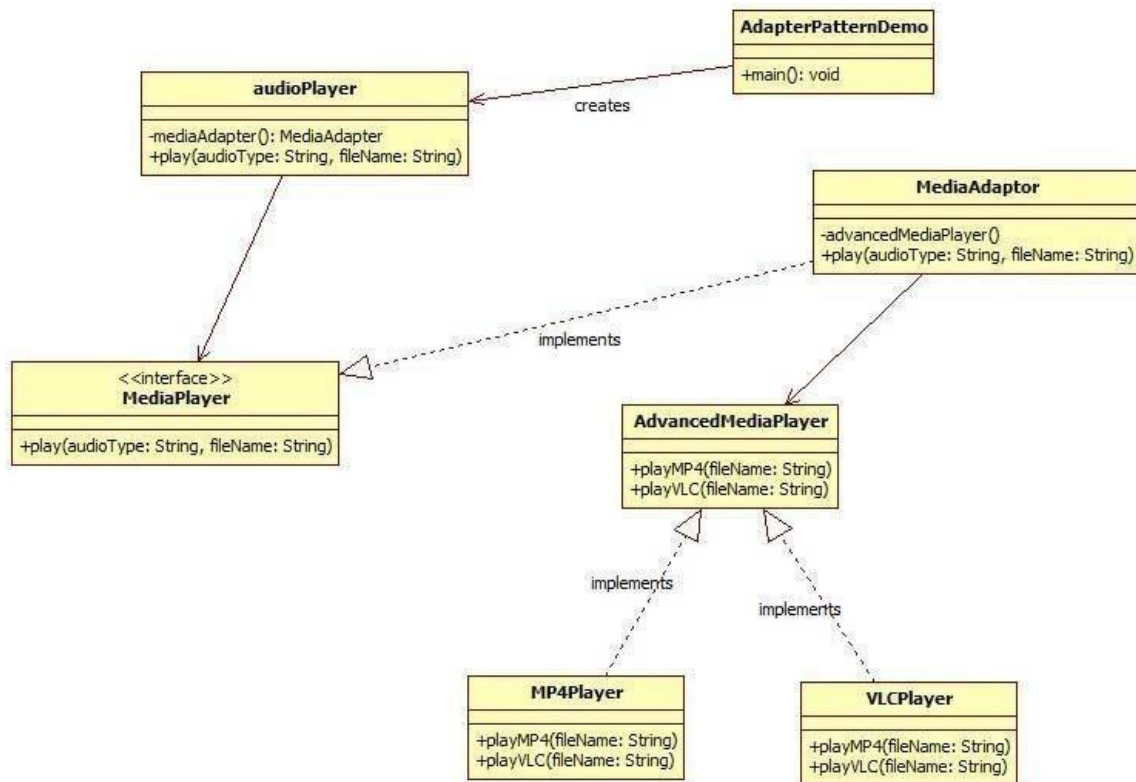


Experiment 6**6. Design and implement an Adapter Pattern for a Music System.****Solution:**

```
interface MediaPlayer {  
  
    void play(String audioType, String fileName);  
  
}  
  
public class AudioPlayer implements MediaPlayer {  
  
    @Override  
  
    public void play(String audioType, String fileName) {  
  
        if(audioType.equalsIgnoreCase("mp3")) {  
  
            System.out.println("Playing MP3 file: " + fileName);  
  
        }  
        else {  
  
            System.out.println("Invalid media format: " + audioType + " not supported");  
  
        }  
    }  
}  
  
class AdapterPatternDemo {  
  
    public static void main(String[] args) {  
  
        AudioPlayer audioPlayer = new AudioPlayer();  
  
        audioPlayer.play("mp3", "song.mp3");  
  
    }  
}
```

Output:

Playing MP3 file: song.mp3



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

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

interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

interface Observer {
    void update(String news);
}

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();
    }
}

class Newspaper implements Observer {
    private String name;

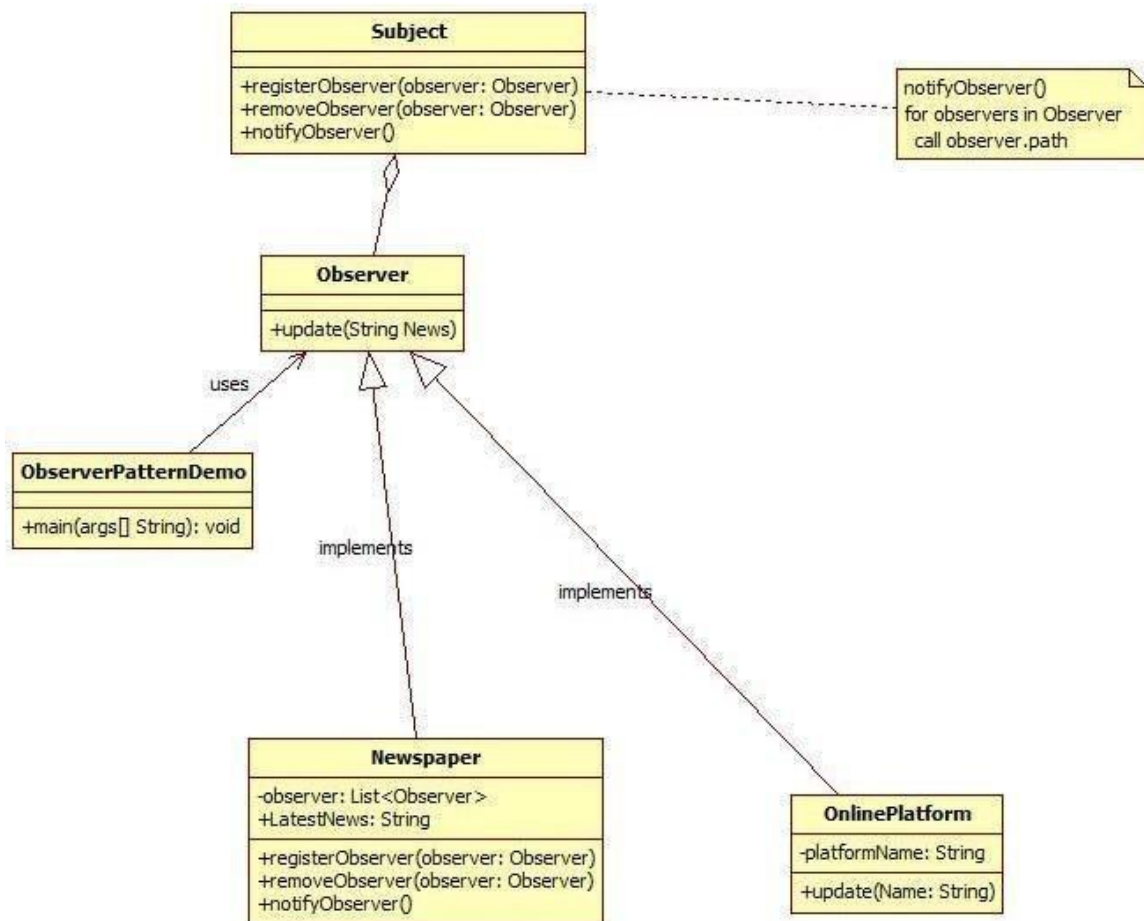
    public Newspaper(String name) {
        this.name = name;
    }

    @Override
    public void update(String news) {
        System.out.println(name + " received news update: " + news);
    }
}

class ObserverPatternDemo {
    public static void main(String[] args) {
        NewsAgency newsAgency = new NewsAgency();
        Observer newspaper1 = new Newspaper("The Daily Times");
        newsAgency.registerObserver(newspaper1);
        newsAgency.setNews("Breaking: New Java Version Released!");
    }
}
```

Output:

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



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

```
class Amplifier {  
    public void setVolume(int level) {  
        System.out.println("Amplifier volume set to " + level);  
    }  
}  
  
class DVDPlayer {  
    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");  
    }  
}  
  
class Projector {  
    public void off() {  
        System.out.println("Projector is off");  
    }  
}  
  
class Screen {  
    public void up() {
```

```
        System.out.println("Screen is going up");
    }
}

class HomeTheaterFacade {

    private Amplifier amplifier;
    private DVDPlayer dvdPlayer;
    private Projector projector;
    private Screen screen;

    public HomeTheaterFacade(Amplifier amp, DVDPlayer dvd, Projector proj, Screen screen) {

        this.amplifier = amp;
        this.dvdPlayer = dvd;
        this.projector = proj;
        this.screen = screen;
    }

    public void watchMovie(String movie) {

        amplifier.setVolume(5);

        dvdPlayer.play(movie);
    }

    public void endMovie() {

        System.out.println("Shutting movie theatre down...");

        screen.up();

        projector.off();
    }
}

class HomeTheaterTestDrive {

    public static void main(String[] args) {

        Amplifier amp = new Amplifier();

        DVDPlayer dvd = new DVDPlayer();

        Projector projector = new Projector();

        Screen screen = new Screen();
    }
}
```

```
HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, dvd, projector, screen);

homeTheater.watchMovie("Inception");

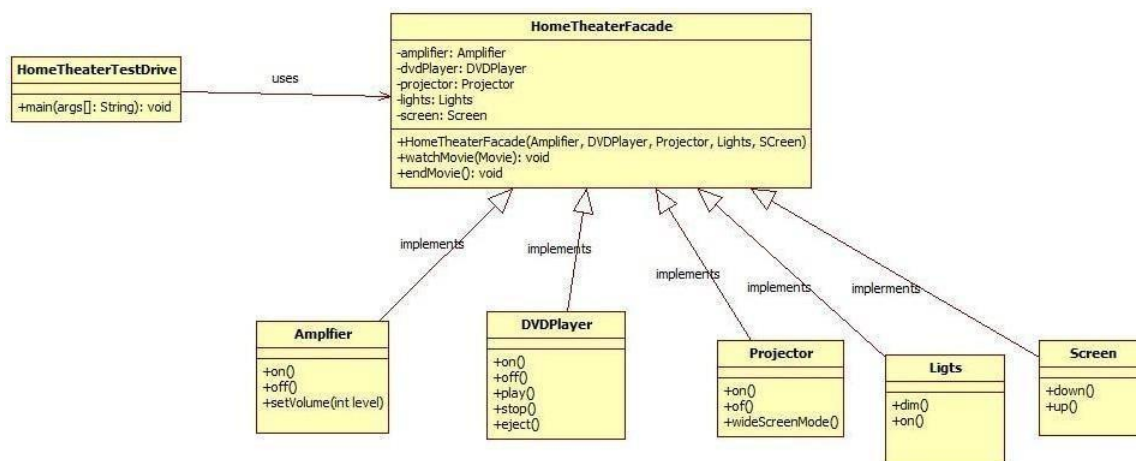
homeTheater.endMovie();

}

}
```

Output:

Amplifier volume set to 5
Playing movie: Inception
Shutting movie theatre down...
Screen is going up
Projector is off



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

```
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();  
}  
  
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...");  
    }  
}
```

```
    }

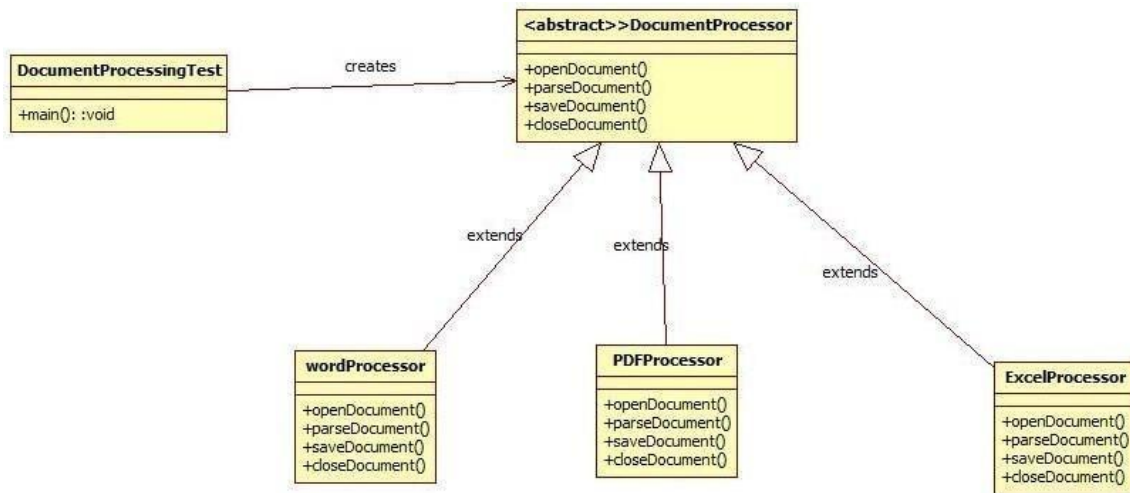
    @Override
    protected void saveDocument() {
        System.out.println("Saving PDF document...");
    }

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

    public static void main(String[] args) {
        // Process a PDF document
        DocumentProcessor pdfProcessor = new PDFProcessor();
        pdfProcessor.processDocument();
    }
}
```

Output:

Opening PDF document... Parsing
PDF document content... Saving
PDF document...
Closing PDF document.



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:

```
import java.util.ArrayList;

import java.util.List;

// Subject interface
interface Subject {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

// Observer interface
interface Observer {

    void update(float temperature, float humidity, float pressure);

}

// WeatherStation class (Subject)

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();

}
}

// CurrentConditionsDisplay class (Observer)

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");
    }
}
```

```
}  
}  
  
// Statistics Display class (Observer)  
  
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);  
  
    }  
}  
  
// WeatherMonitoringSystemTest (main class)  
  
class WeatherMonitoringSystemTest {  
  
    public static void main(String[] args) {  
  
        // Create WeatherStation (Subject)  
        WeatherStation weatherStation = new WeatherStation();  
  
        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();
```

```

StatisticsDisplay statisticsDisplay = new StatisticsDisplay();

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

weatherStation.registerObserver(statisticsDisplay);

// Simulate new weather measurements

weatherStation.setWeatherData(25.5f, 65.0f, 30.4f); // First update

}

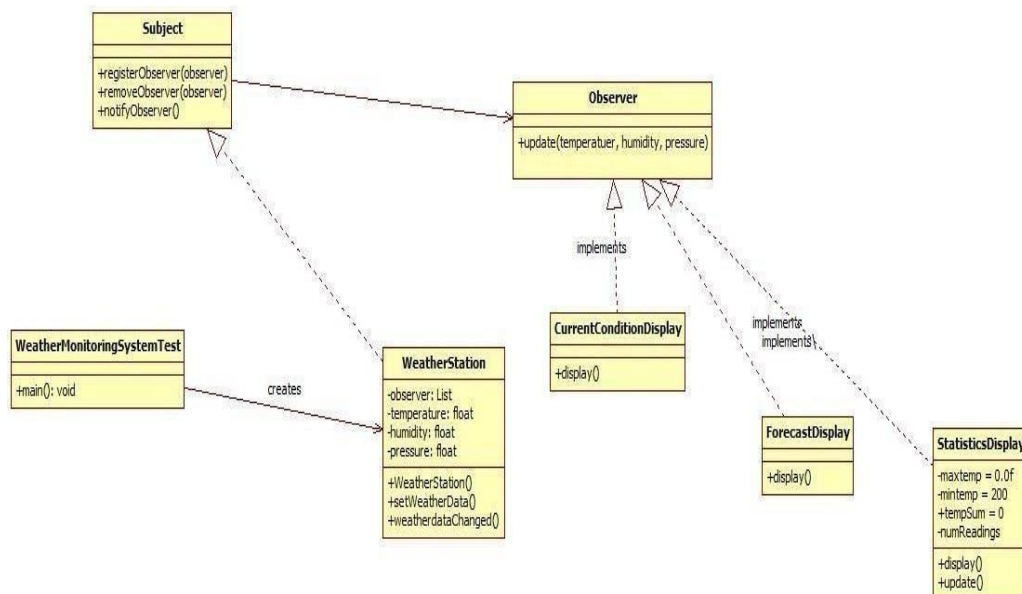
}

```

Output:

Current conditions: 25.5C degrees and 65.0% humidity

Avg/Max/Min temperature = 25.5/25.5/25.5



Experiment 11

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

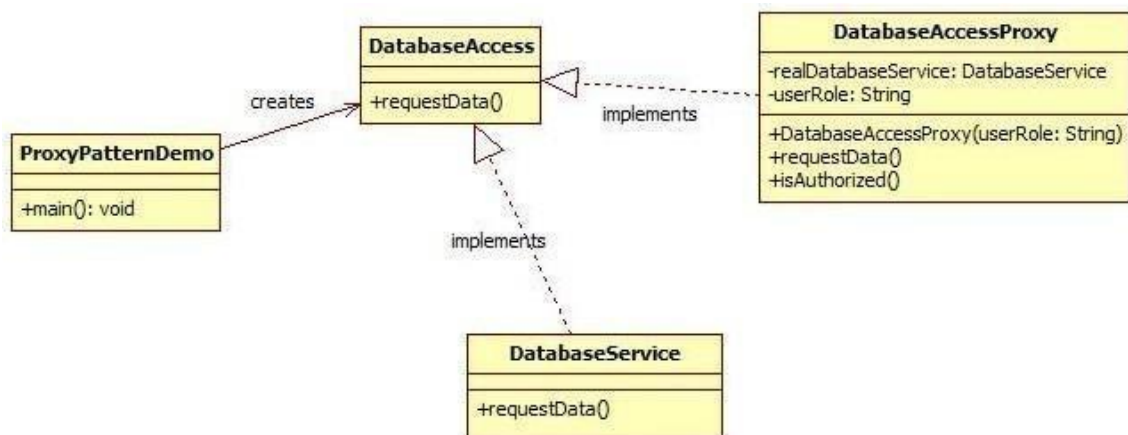
Solution:

```
interface DatabaseAccess {  
  
    void requestData();  
  
}  
  
class DatabaseService implements DatabaseAccess {  
  
    @Override  
    public void requestData() {  
  
        System.out.println("Fetching data from the database...");  
  
    }  
}  
  
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();  
  
        }  
    }  
  
    private boolean isAuthorized() {  
  
        return "ADMIN".equals(userRole);  
  
    }  
}
```

```
}  
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();  
  
    }  
}
```

Output:

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



Experiment 12

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

Solution:

```
import java.util.ArrayList;

import java.util.List;

interface ChatRoomMediator {

    void sendMessage(String message, Participant participant);

    void addParticipant(Participant participant);

}

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);

    }

}
```

```
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);
}

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);

    }
}

class MediatorPatternDemo {

    public static void main(String[] args) {

        // Create the mediator (Chat Room)

        ChatRoomMediator chatRoom = new ConcreteChatRoom();
```

```

// Create participants (only Alice and Bob)

Participant user1 = new UserParticipant("Alice", chatRoom);

Participant user2 = new UserParticipant("Bob", chatRoom);

// Add participants to the chat room

chatRoom.addParticipant(user1);

chatRoom.addParticipant(user2);

// Alice sends a message

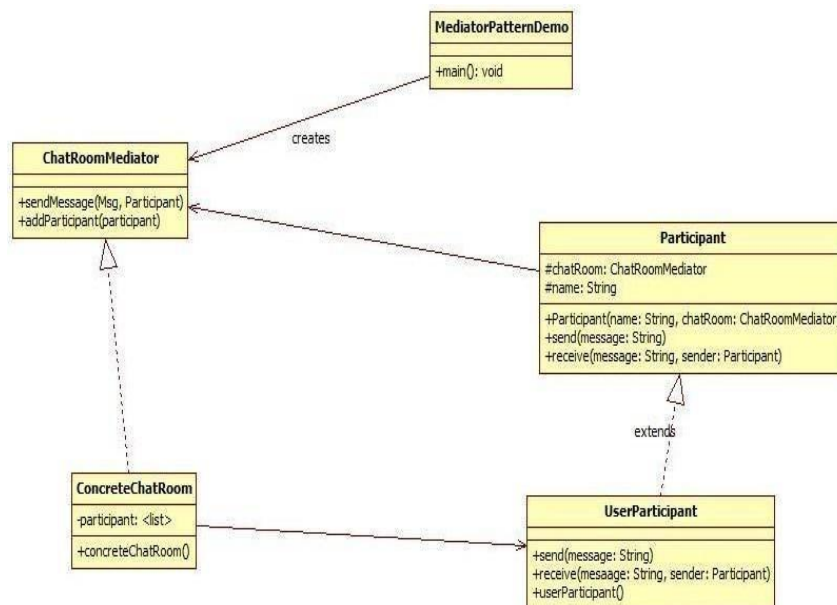
user1.send("Hello, everyone!");
}
}

```

Output:

Alice sends: Hello, everyone!

Bob receives a message from Alice: Hello, everyone!



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?