

DESIGN PRINCIPLES AND PATTERNS MANDATORY

Superset ID : 6384831

Name : Mohana Priya N

E-mail : mohanapriya.2205056@srec.ac.in

Mandatory Questions:

1) Exercise 1: Implementing the Singleton Pattern

Solution:

//Logger.java

```
package palindrome;

public class Logger {
    private static Logger instance;
    private Logger() {
        System.out.println("Logger Initialized");
    }
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance; }
    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

//Main.java

```
package palindrome;

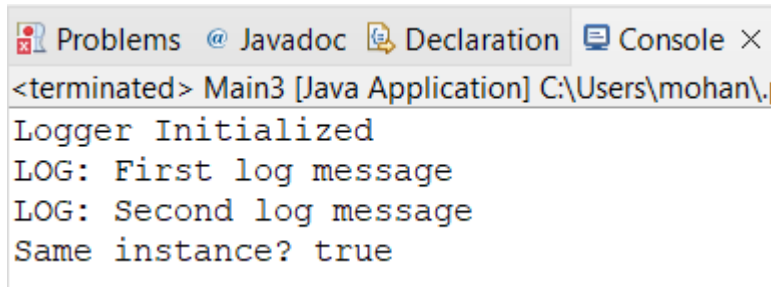
public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();
        logger1.log("First log message");
    }
}
```

```

        logger2.log("Second log message");
        System.out.println("Same instance? " + (logger1 == logger2));
    }
}

```

Output:



```

<terminated> Main3 [Java Application] C:\Users\mohan\...
Logger Initialized
LOG: First log message
LOG: Second log message
Same instance? true

```

2) Exercise 2: Implementing the Factory Method Pattern

Solution:

//TestFactory.java

```

package palindrome;

interface Document {
    void open();
}

class WordDocument implements Document {
    public void open() {
        System.out.println("Opening Word Document");
    }
}

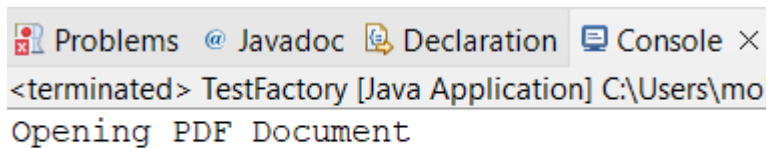
class PdfDocument implements Document {
    public void open() {
        System.out.println("Opening PDF Document");
    }
}

class ExcelDocument implements Document {
    public void open() {

```

```
        System.out.println("Opening Excel Document");
    }
}
abstract class DocumentFactory {
    abstract Document createDocument();
}
class WordFactory extends DocumentFactory {
    Document createDocument() {
        return new WordDocument();
    }
}
class PdfFactory extends DocumentFactory {
    Document createDocument() {
        return new PdfDocument();
    }
}
public class TestFactory {
    public static void main(String[] args) {
        DocumentFactory factory = new PdfFactory();
        Document doc = factory.createDocument();
        doc.open();
    }
}
```

Output:



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the Java application. The output consists of two lines: "<terminated> TestFactory [Java Application] C:\Users\mo" and "Opening PDF Document".

```
<terminated> TestFactory [Java Application] C:\Users\mo
Opening PDF Document
```

Other Questions:

3) Exercise 3: Implementing the Builder Pattern

Solution:

//Computer.java

```
package palindrome;

public class Computer {
    private String CPU;
    private String RAM;
    private String Storage;
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.Storage = builder.Storage;
    }
    public static class Builder {
        private String CPU;
        private String RAM;
        private String Storage;
        public Builder setCPU(String cpu) {
            this.CPU = cpu;
            return this;
        }
        public Builder setRAM(String ram) {
            this.RAM = ram;
            return this;
        }
        public Builder setStorage(String storage) {
            this.Storage = storage;
            return this;
        }
        public Computer build() {
```

```

        return new Computer(this);
    }
}

public void showSpecs() {
    System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + Storage);
}
}

```

//TestBuilder.java

```

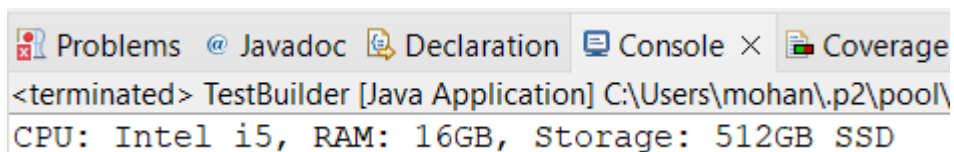
package palindrome;

public class TestBuilder {

    public static void main(String[] args) {
        Computer myComputer = new Computer.Builder()
            .setCPU("Intel i5")
            .setRAM("16GB")
            .setStorage("512GB SSD")
            .build();
        myComputer.showSpecs();
    }
}

```

Output:



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, Console, and Coverage. The Console tab is active, displaying the output of the TestBuilder application. The output is: <terminated> TestBuilder [Java Application] C:\Users\mohan\.p2\pool\ CPU: Intel i5, RAM: 16GB, Storage: 512GB SSD

4) Exercise 4: Implementing the Adapter Pattern

Solution:

// PaymentProcessor.java

```

interface PaymentProcessor {
    void processPayment();
}

```

// ThirdPartyGateway.java

```
class ThirdPartyGateway {  
    void makeTransaction() {  
        System.out.println("Processing payment with third-party gateway...");  
    }  
}
```

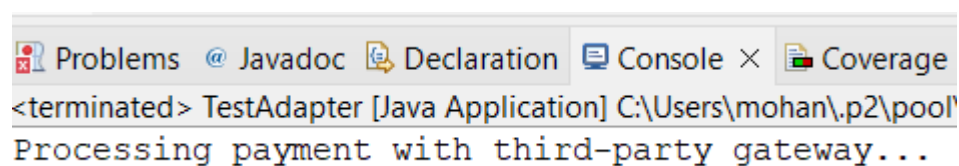
// GatewayAdapter.java

```
class GatewayAdapter implements PaymentProcessor {  
    private ThirdPartyGateway gateway;  
    public GatewayAdapter(ThirdPartyGateway gateway) {  
        this.gateway = gateway;  
    }  
    public void processPayment() {  
        gateway.makeTransaction();  
    }  
}
```

// TestAdapter.java

```
public class TestAdapter {  
    public static void main(String[] args) {  
        ThirdPartyGateway oldGateway = new ThirdPartyGateway();  
        PaymentProcessor processor = new GatewayAdapter(oldGateway);  
        processor.processPayment();  
    }  
}
```

Output:



The screenshot shows an IDE interface with tabs for Problems, Javadoc, Declaration, Console, and Coverage. The Console tab is active, displaying the output of the Java application: `<terminated> TestAdapter [Java Application] C:\Users\mohan\.p2\pool'` followed by `Processing payment with third-party gateway...` on a new line.

5) Exercise 5: Implementing the Decorator Pattern

Solution:

// Notifier.java

```
interface Notifier {  
    void send(String message);  
}
```

// EmailNotifier.java

```
class EmailNotifier implements Notifier {  
    public void send(String message) {  
        System.out.println("Sending Email: " + message);  
    }  
}
```

// NotifierDecorator.java

```
abstract class NotifierDecorator implements Notifier {  
    protected Notifier notifier;  
    public NotifierDecorator(Notifier notifier) {  
        this.notifier = notifier;  
    }  
    public void send(String message) {  
        notifier.send(message);  
    }  
}
```

// SMSNotifierDecorator.java

```
class SMSNotifierDecorator extends NotifierDecorator {  
    public SMSNotifierDecorator(Notifier notifier) {  
        super(notifier);  
    }  
}
```

```

    public void send(String message) {
        super.send(message);
        System.out.println("Sending SMS: " + message);
    }
}

```

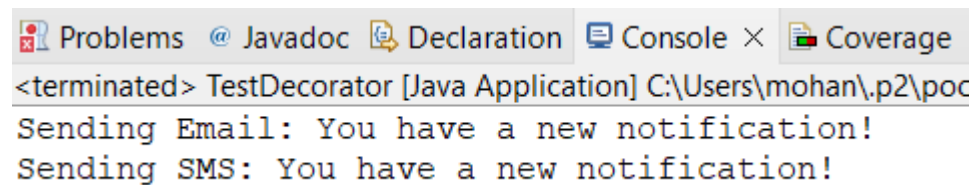
// TestDecorator.java

```

public class TestDecorator {
    public static void main(String[] args) {
        Notifier notifier = new SMSNotifierDecorator(new EmailNotifier());
        notifier.send("You have a new notification!");
    }
}

```

Output:



```

<terminated> TestDecorator [Java Application] C:\Users\mohan\.p2\poc
Sending Email: You have a new notification!
Sending SMS: You have a new notification!

```

6) Exercise 6: Implementing the Proxy Pattern

Solution:

// Image.java

```

interface Image {
    void display();
}

```

// RealImage.java

```

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();}
}

```



```
private void loadFromDisk() {  
    System.out.println("Loading image: " + filename);  
}  
public void display() {  
    System.out.println("Displaying: " + filename);  
}  
}
```

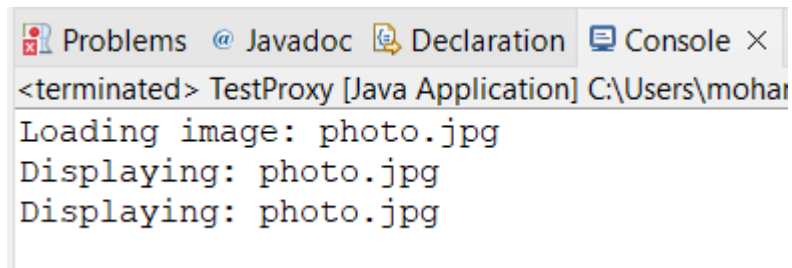
// ProxyImage.java

```
class ProxyImage implements Image {  
    private RealImage realImage;  
    private String filename;  
    public ProxyImage(String filename) {  
        this.filename = filename;  
    }  
    public void display() {  
        if (realImage == null) {  
            realImage = new RealImage(filename);  
        }  
        realImage.display();  
    }  
}
```

// TestProxy.java

```
public class TestProxy {  
    public static void main(String[] args) {  
        Image img = new ProxyImage("photo.jpg");  
        img.display();  
        img.display();  
    }  
}
```

Output:



```
<terminated> TestProxy [Java Application] C:\Users\mohar
Loading image: photo.jpg
Displaying: photo.jpg
Displaying: photo.jpg
```

7) Exercise 7: Implementing the Observer Pattern

Solution:

// Observer.java

```
interface Observer {
    void update(float price);
}
```

// Stock.java

```
interface Stock {
    void register(Observer o);
    void deregister(Observer o);
    void notifyObservers();
}
```

// StockMarket.java

```
import java.util.*;

class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private float price;

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

    public void register(Observer o) {
```

```
        observers.add(o);
    }

    public void deregister(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(price);
        }
    }
}
```

// MobileApp.java

```
class MobileApp implements Observer {
    public void update(float price) {
        System.out.println("Mobile App - New Stock Price: " + price);
    }
}
```

// WebApp.java

```
class WebApp implements Observer {
    public void update(float price) {
        System.out.println("Web App - New Stock Price: " + price);
    }
}
```

// TestObserver.java

```
public class TestObserver {
    public static void main(String[] args) {
        StockMarket market = new StockMarket();
    }
}
```

```

    Observer mobile = new MobileApp();

    Observer web = new WebApp();

    market.register(mobile);

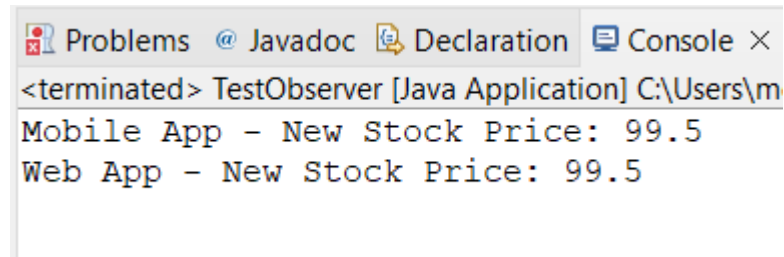
    market.register(web);

    market.setPrice(99.5f);

}
}

```

Output:



```

<terminated> TestObserver [Java Application] C:\Users\m\n\n
Mobile App - New Stock Price: 99.5
Web App - New Stock Price: 99.5

```

8) Exercise 8: Implementing the Strategy Pattern

Solution:

// PaymentStrategy.java

```

interface PaymentStrategy {

    void pay(int amount);

}

```

// CreditCardPayment.java

```

class CreditCardPayment implements PaymentStrategy {

    public void pay(int amount) {

        System.out.println("Paid " + amount + " using Credit Card.");

    }

}

```

// PayPalPayment.java

```

class PayPalPayment implements PaymentStrategy {

    public void pay(int amount) {


```

```
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

// **PaymentContext.java**

```
class PaymentContext {
    private PaymentStrategy strategy;

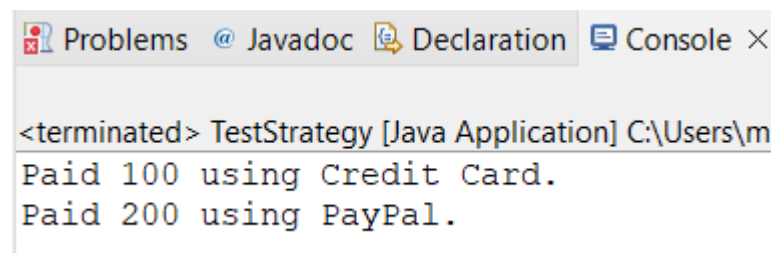
    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void pay(int amount) {
        strategy.pay(amount);
    }
}
```

// **TestStrategy.java**

```
public class TestStrategy {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();
        context.setStrategy(new CreditCardPayment());
        context.pay(100);
        context.setStrategy(new PayPalPayment());
        context.pay(200);
    }
}
```

Output:



The screenshot shows an IDE window with tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of the 'TestStrategy' application. The output consists of two lines: 'Paid 100 using Credit Card.' and 'Paid 200 using PayPal.'.

```
<terminated> TestStrategy [Java Application] C:\Users\m
Paid 100 using Credit Card.
Paid 200 using PayPal.
```

9) Exercise 9: Implementing the Command Pattern

Solution:

// Command.java

```
interface Command {  
    void execute();  
}
```

// Light.java

```
class Light {  
    void turnOn() {  
        System.out.println("Light is ON");  
    }  
    void turnOff() {  
        System.out.println("Light is OFF");  
    }  
}
```

// LightOnCommand.java

```
class LightOnCommand implements Command {  
    Light light;  
    LightOnCommand(Light light) {  
        this.light = light;  
    }  
    public void execute() {  
        light.turnOn();  
    }  
}
```

// LightOffCommand.java

```
class LightOffCommand implements Command {
```

```
Light light;

LightOffCommand(Light light) {
    this.light = light;
}

public void execute() {
    light.turnOff();
}
}
```

// RemoteControl.java

```
class RemoteControl {
    Command command;

    void setCommand(Command command) {
        this.command = command;
    }

    void pressButton() {
        command.execute();
    }
}
```

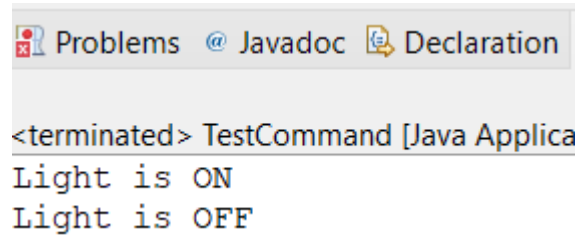
// TestCommand.java

```
public class TestCommand {

    public static void main(String[] args) {
        Light light = new Light();
        Command on = new LightOnCommand(light);
        Command off = new LightOffCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.setCommand(on);
        remote.pressButton();
        remote.setCommand(off);
    }
}
```

```
        remote.pressButton();
    }
}
```

Output:



```
<terminated> TestCommand [Java Applica
Light is ON
Light is OFF
```

10) Exercise 10: Implementing the MVC Pattern

Solution:

// Student.java

```
class Student {
    private String name;
    private String id;
    private String grade;
    public String getName() { return name; }
    public String getId() { return id; }
    public String getGrade() { return grade; }
    public void setName(String name) { this.name = name; }
    public void setId(String id) { this.id = id; }
    public void setGrade(String grade) { this.grade = grade; }
}
```

// StudentView.java

```
class StudentView {
    public void displayStudentDetails(Student student) {
        System.out.println("Student ID: " + student.getId());
        System.out.println("Name: " + student.getName());
        System.out.println("Grade: " + student.getGrade());
    }
}
```



```
}  
}
```

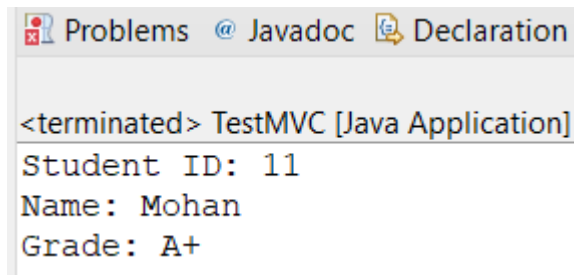
// StudentController.java

```
class StudentController {  
    private Student student;  
    private StudentView view;  
    public StudentController(Student student, StudentView view) {  
        this.student = student;  
        this.view = view;  
    }  
    public void updateView() {  
        view.displayStudentDetails(student);  
    }  
}
```

// TestMVC.java

```
public class TestMVC {  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.setId("11");  
        student.setName("Mohan");  
        student.setGrade("A+");  
        StudentView view = new StudentView();  
        StudentController controller = new StudentController(student, view);  
        controller.updateView();  
    }  
}
```

Output:

A screenshot of an IDE's output window. The title bar shows 'Problems', '@ Javadoc', and 'Declaration'. The main content area displays the output of a Java application: '<terminated> TestMVC [Java Application]', 'Student ID: 11', 'Name: Mohan', and 'Grade: A+'.

```
<terminated> TestMVC [Java Application]
Student ID: 11
Name: Mohan
Grade: A+
```

11) Exercise 11: Implementing Dependency Injection

Solution:

// CustomerRepository.java

```
interface CustomerRepository {
    String findCustomerById(String id);
}
```

// CustomerRepositoryImpl.java

```
class CustomerRepositoryImpl implements CustomerRepository {
    public String findCustomerById(String id) {
        return "Customer with ID: " + id;
    }
}
```

// CustomerService.java

```
class CustomerService {
    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }

    public void showCustomer(String id) {
        System.out.println(repository.findCustomerById(id));
    }
}
```

// TestDI.java

```
public class TestDI {  
    public static void main(String[] args) {  
        CustomerRepository repo = new CustomerRepositoryImpl();  
        CustomerService service = new CustomerService(repo);  
        service.showCustomer("C101");  
    }  
}
```

Output:

