Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern

Create a folder: SingletonPatternExample

Inside it, create two files:

Logger.java

```java
public class Logger {
    private static Logger instance;

    private Logger() {
        System.out.println(x:"Logger instance created.");
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

TestLogger.java

```java
public class TestLogger {
    Run | Debug
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        logger1.log(message:"First log message");

        Logger logger2 = Logger.getInstance();
        logger2.log(message:"Second log message");

        if (logger1 == logger2) {
            System.out.println(x:"Both logger1 and logger2 refer to the same instance.");
        } else {
            System.out.println(x:"Different instances were created! Singleton failed.");
        }
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\SingletonPatternExample> java TestLogger
>>
Logger instance created.
Log: First log message
Log: Second log message
Both logger1 and logger2 refer to the same instance.
```

Exercise 2: Implementing the Factory Method Pattern

Create a folder: FactoryMethodPatternExample

Inside it, create 9 files:

Document.java

```java
public interface Document {
    void open();
}
```

WordDocument.java

```java
public class WordDocument implements Document {
    public void open() {
        System.out.println(x:"Opening a Word document.");
    }
}
```

PdfDocument.java

```java
public class PdfDocument implements Document {
    public void open() {
        System.out.println(x:"Opening a PDF document.");
    }
}
```

ExcelDocument.java

```java
public class ExcelDocument implements Document {
    public void open() {
        System.out.println(x:"Opening an Excel document.");
    }
}
```

DocumentFactory.java

```java
public abstract class DocumentFactory {
    public abstract Document createDocument();
}
```

WordFactory.java

```java
public class WordFactory extends DocumentFactory {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

PdfFactory.java

```java
public class PdfFactory extends DocumentFactory {
    public Document createDocument() {
        return new PdfDocument();
    }
}
```

ExcelFactory.java

```java
public class ExcelFactory extends DocumentFactory {
    public Document createDocument() {
        return new ExcelDocument();
    }
}
```

TestFactoryPattern.java

```java
public class TestFactoryPattern {
    Run | Debug
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();

        DocumentFactory pdfFactory = new PdfFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\FactoryMethodPatternExample> javac *.java
>> java TestFactoryPattern
>>
Opening a Word document.
Opening a PDF document.
Opening an Excel document.
```

Exercise 3: Implementing the Builder Pattern

Create a folder: BuilderPatternExample

Inside it, create two files:

Computer.java

```java
public class Computer {
    // Required attributes
    private final String CPU;
    private final String RAM;

    // Optional attributes
    private final String storage;
    private final String graphicsCard;
    private final String operatingSystem;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.graphicsCard = builder.graphicsCard;
        this.operatingSystem = builder.operatingSystem;
    }

    // Static nested Builder class
    public static class Builder {
        private final String CPU;
        private final String RAM;

        private String storage;
        private String graphicsCard;
        private String operatingSystem;

        public Builder(String CPU, String RAM) {
            this.CPU = CPU;
            this.RAM = RAM;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public Builder setGraphicsCard(String graphicsCard) {
            this.graphicsCard = graphicsCard;
            return this;
        }

        public Builder setOperatingSystem(String os) {
            this.operatingSystem = os;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    // ToString for displaying the configuration
    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +
                ", GraphicsCard=" + graphicsCard + ", OS=" + operatingSystem + "]";
    }
}
```

TestBuilderPattern.java

```java
public class TestBuilderPattern {
    Run | Debug
    public static void main(String[] args) {
        // Basic configuration
        Computer basicComputer = new Computer.Builder(CPU:"Intel i3", RAM:"4GB").build();

        // Gaming configuration
        Computer gamingComputer = new Computer.Builder(CPU:"Intel i9", RAM:"32GB")
                .setStorage(storage:"1TB SSD")
                .setGraphicsCard(graphicsCard:"NVIDIA RTX 4090")
                .setOperatingSystem(os:"Windows 11 Pro")
                .build();

        // Developer configuration
        Computer devComputer = new Computer.Builder(CPU:"AMD Ryzen 7", RAM:"16GB")
                .setStorage(storage:"512GB SSD")
                .setOperatingSystem(os:"Ubuntu Linux")
                .build();

        System.out.println("Basic Config: " + basicComputer);
        System.out.println("Gaming Config: " + gamingComputer);
        System.out.println("Developer Config: " + devComputer);
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\BuilderPatternExample> javac Computer.java TestBuilderPattern.java
>> java TestBuilderPattern
>>
Basic Config: Computer [CPU=Intel i3, RAM=4GB, Storage=null, GraphicsCard=null, OS=null]
Gaming Config: Computer [CPU=Intel i9, RAM=32GB, Storage=1TB SSD, GraphicsCard=NVIDIA RTX 4090, OS=Windows 11 Pro]
Developer Config: Computer [CPU=AMD Ryzen 7, RAM=16GB, Storage=512GB SSD, GraphicsCard=null, OS=Ubuntu Linux]
```

Exercise 4: Implementing the Adapter Pattern

Create a folder: AdapterPatternExample

Inside it, create 6 files:

PaymentProcessor.java

```java
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

PayPalGateway.java

```java
public class PayPalGateway {
    public void makePayPalPayment(double amount) {
        System.out.println("Processing payment through PayPal: $" + amount);
    }
}
```

StripeGateway.java

```java
public class StripeGateway {
    public void sendStripePayment(double amount) {
        System.out.println("Processing payment through Stripe: $" + amount);
    }
}
```

PayPalAdapter.java

```java
public class PayPalAdapter implements PaymentProcessor {
    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway) {
        this.payPalGateway = payPalGateway;
    }

    public void processPayment(double amount) {
        payPalGateway.makePayPalPayment(amount);
    }
}
```

StripeAdapter.java

```java
public class StripeAdapter implements PaymentProcessor {
    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway stripeGateway) {
        this.stripeGateway = stripeGateway;
    }

    public void processPayment(double amount) {
        stripeGateway.sendStripePayment(amount);
    }
}
```

TestAdapterPattern.java

```java
public class TestAdapterPattern {
    Run | Debug
    public static void main(String[] args) {
        // Using PayPal
        PaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());
        paypal.processPayment(amount:250.00);

        // Using Stripe
        PaymentProcessor stripe = new StripeAdapter(new StripeGateway());
        stripe.processPayment(amount:450.50);
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\AdapterPatternExample> javac *.java
>> java TestAdapterPattern
>>
Processing payment through PayPal: $250.0
Processing payment through Stripe: $450.5
PS C:\Codes\DIgital Nurture\DSA\AdapterPatternExample>
```

Exercise 5: Implementing the Decorator Pattern

Create a folder: DecoratorPatternExample

Inside it, create 6 files:

Notifier.java

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

EmailNotifier.java

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

NotifierDecorator.java

```java
public abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {
        this.notifier = notifier;
    }

    public void send(String message) {
        notifier.send(message); // delegate to base notifier
    }
}
```

SMSDecorator.java

```java
public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    public void send(String message) {
        super.send(message);
        sendSMS(message);
    }

    private void sendSMS(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```

SlackNotifierDecorator.java

```java
public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    public void send(String message) {
        super.send(message);
        sendSlack(message);
    }

    private void sendSlack(String message) {
        System.out.println("Sending Slack message: " + message);
    }
}
```

TestDecoratorPattern.java

```java
public class TestDecoratorPattern {
    Run | Debug
    public static void main(String[] args) {
        // Step-by-step wrapping: Email -> SMS -> Slack
        Notifier notifier = new SlackNotifierDecorator(
                                new SMSNotifierDecorator(
                                    new EmailNotifier()
                                )
                            );

        notifier.send(message:"System maintenance at 2 AM.");
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\DecoratorPatternExample> javac *.java
>> java TestDecoratorPattern
>>
Sending Email: System maintenance at 2 AM.
Sending SMS: System maintenance at 2 AM.
Sending Slack message: System maintenance at 2 AM.
```

Exercise 6: Implementing the Proxy Pattern

Create a folder: ProxyPatternExample

Inside it, create 4 files:

Image.java

```java
public interface Image {
    void display();
}
```

RealImage.java

```java
public class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromRemoteServer();
    }

    private void loadFromRemoteServer() {
        System.out.println("Loading image from remote server: " + filename);
    }

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

ProxyImage.java

```java
public class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);  // Lazy loading
        }
        realImage.display();
    }
}
```

TestProxyPattern.java

```java
public class TestProxyPattern {
    Run | Debug
    public static void main(String[] args) {
        Image image1 = new ProxyImage(filename:"nature_photo.jpg");
        Image image2 = new ProxyImage(filename:"mountain_view.jpg");

        // Image is loaded only when display is called
        System.out.println(x:"First time displaying image1:");
        image1.display();  // Loads and displays

        System.out.println(x:"\nSecond time displaying image1:");
        image1.display();  // Just displays (cached)

        System.out.println(x:"\nDisplaying image2:");
        image2.display();  // Loads and displays
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\ProxyPatternExample> javac *.java
>> java TestProxyPattern
>>
First time displaying image1:
Loading image from remote server: nature_photo.jpg
Displaying: nature_photo.jpg

Second time displaying image1:
Displaying: nature_photo.jpg

Displaying image2:
Loading image from remote server: mountain_view.jpg
Displaying: mountain_view.jpg
PS C:\Codes\DIgital Nurture\DSA\ProxyPatternExample>
```

Exercise 7: Implementing the Observer Pattern

Create a folder: ObserverPatternExample

Inside it, create 6 files:

Stock.java

```java
public interface Stock {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

StockMarket.java

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

public class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private String stockName;
    private double stockPrice;

    public void setStockPrice(String stockName, double stockPrice) {
        this.stockName = stockName;
        this.stockPrice = stockPrice;
        notifyObservers();
    }

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

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

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(stockName, stockPrice);
        }
    }
}
```

Observer.java

```java
public interface Observer {
    void update(String stockName, double price);
}
```

MobileApp.java

```java
public class MobileApp implements Observer {
    private String name;

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

    public void update(String stockName, double price) {
        System.out.println("Mobile App [" + name + "] - Stock: " + stockName + " updated to $" + price);
    }
}
```

WebApp.java

```java
public class WebApp implements Observer {
    private String name;

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

    public void update(String stockName, double price) {
        System.out.println("Web App [" + name + "] - Stock: " + stockName + " updated to $" + price);
    }
}
```

TestObserverPattern.java

```java
public class TestObserverPattern {
    Run | Debug
    public static void main(String[] args) {
        StockMarket market = new StockMarket();

        Observer mobileUser = new MobileApp(name:"Alice");
        Observer webUser = new WebApp(name:"Bob");

        market.registerObserver(mobileUser);
        market.registerObserver(webUser);

        market.setStockPrice(stockName:"AAPL", stockPrice:182.15);
        market.setStockPrice(stockName:"GOOGL", stockPrice:2780.30);

        market.removeObserver(mobileUser);

        market.setStockPrice(stockName:"AAPL", stockPrice:190.00);
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\ObserverPatternExample> jav
>> java TestObserverPattern
>>
Mobile App [Alice] - Stock: AAPL updated to $182.15
Web App [Bob] - Stock: AAPL updated to $182.15
Mobile App [Alice] - Stock: GOOGL updated to $2780.3
Web App [Bob] - Stock: GOOGL updated to $2780.3
Web App [Bob] - Stock: AAPL updated to $190.0
```

Exercise 8: Implementing the Strategy Pattern

Create a folder: StrategyPatternExample

Inside it, create 5 files:

PaymentStrategy.java

```java
public interface PaymentStrategy {
    void pay(double amount);
}
```

CreditCardPayment.java

```java
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));
    }
}
```

PayPalPayment.java

```java
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal account: " + email);
    }
}
```

PaymentContext.java

```java
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

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

    public void pay(double amount) {
        if (paymentStrategy == null) {
            System.out.println(x:"Payment strategy not set.");
        } else {
            paymentStrategy.pay(amount);
        }
    }
}
```

TestStrategyPattern.java

```java
public class TestStrategyPattern {
    Run | Debug
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        // Use Credit Card payment
        context.setPaymentStrategy(new CreditCardPayment(cardNumber:"1234567890123456"));
        context.pay(amount:150.0);

        // Use PayPal payment
        context.setPaymentStrategy(new PayPalPayment(email:"user@example.com"));
        context.pay(amount:75.5);
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\StrategyPatternExample> javac *.java
>> java TestStrategyPattern
>>
Paid $150.0 using Credit Card ending with 3456
Paid $75.5 using PayPal account: user@example.com
```

Exercise 9: Implementing the Command Pattern

Create a folder: CommandPatternExample

Inside it, create 6 files:

Command.java

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

Light.java

```java
public class Light {
    public void turnOn() {
        System.out.println(x:"The light is ON");
    }

    public void turnOff() {
        System.out.println(x:"The light is OFF");
    }
}
```

LightOnCommand.java

```java
public class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

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

LightOffCommand.java

```java
public class LightOffCommand implements Command {
    private Light light;

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

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

RemoteControl.java

```java
public class RemoteControl {
    private Command command;

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

    public void pressButton() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println(x:"No command set.");
        }
    }
}
```

TestCommandPattern.java

```java
public class TestCommandPattern {
    Run | Debug
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        System.out.println(x:"Pressing ON button:");
        remote.setCommand(lightOn);
        remote.pressButton();

        System.out.println(x:"Pressing OFF button:");
        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\CommandPatternExample> javac *.java
>> java TestCommandPattern
>>
Pressing ON button:
The light is ON
Pressing OFF button:
The light is OFF
```

Exercise 10: Implementing the MVC Pattern

Create a folder: MVCPatternExample

Inside it, create 4 files:

Student.java

```java
public class Student {
    private String name;
    private String id;
    private String grade;

    public Student(String name, String id, String grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
    }

    // Getters and setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

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

    public String getGrade() { return grade; }
    public void setGrade(String grade) { this.grade = grade; }
}
```

StudentView.java

```java
public class StudentView {
    public StudentView() {
    }

    public void displayStudentDetails(String var1, String var2, String var3) {
        System.out.println("=== Student Details ===");
        System.out.println("Name : " + var1);
        System.out.println("ID   : " + var2);
        System.out.println("Grade: " + var3);
    }
}
```

StudentController.java

```java
public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name) { model.setName(name); }
    public void setStudentId(String id) { model.setId(id); }
    public void setStudentGrade(String grade) { model.setGrade(grade); }

    public String getStudentName() { return model.getName(); }
    public String getStudentId() { return model.getId(); }
    public String getStudentGrade() { return model.getGrade(); }

    public void updateView() {                    String Student.getName()
        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());
    }
}
```

MVCPatternDemo.java

```java
public class MVCPatternDemo {
    Run | Debug
    public static void main(String[] args) {
        // Create model and view
        Student student = new Student(name:"John Doe", id:"S101", grade:"A");
        StudentView view = new StudentView();

        // Create controller
        StudentController controller = new StudentController(student, view);

        // Initial view
        controller.updateView();

        // Update model data via controller
        controller.setStudentName(name:"Alice Smith");
        controller.setStudentGrade(grade:"A+");

        // Updated view
        System.out.println(x:"\nAfter updating student details:");
        controller.updateView();
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\MVCPatternExample> javac *.java
>> java MVCPatternDemo
>>
=== Student Details ===
Name : John Doe
ID   : S101
Grade: A

After updating student details:
=== Student Details ===
Name : Alice Smith
ID   : S101
Grade: A+
```

Exercise 11: Implementing Dependency Injection

Create a folder: DependencyInjectionExample

Inside it, create 5 files:

Customer.java

```java
public class Customer {
    private String id;
    private String name;

    public Customer(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() { return id; }
    public String getName() { return name; }
}
```

CustomerRepository.java

```java
public interface CustomerRepository {
    Customer findCustomerById(String id);
}
```

CustomerRepositoryImpl.java

```java
import java.util.HashMap;
import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {
    private Map<String, Customer> customers = new HashMap<>();

    public CustomerRepositoryImpl() {
        // Add sample customers
        customers.put(key:"C001", new Customer(id:"C001", name:"John Doe"));
        customers.put(key:"C002", new Customer(id:"C002", name:"Alice Smith"));
    }

    public Customer findCustomerById(String id) {
        return customers.get(id);
    }
}
```

CustomerService.java

```java
public class CustomerService {
    private CustomerRepository customerRepository;

    // Constructor injection
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void getCustomerInfo(String id) {
        Customer customer = customerRepository.findCustomerById(id);
        if (customer != null) {
            System.out.println("Customer ID: " + customer.getId());
            System.out.println("Customer Name: " + customer.getName());
        } else {
            System.out.println(x:"Customer not found.");
        }
    }
}
```

DependencyInjectionDemo.java

```java
public class DependencyInjectionDemo {
    Run | Debug
    public static void main(String[] args) {
        // Create repository
        CustomerRepository repository = new CustomerRepositoryImpl();

        // Inject repository into service
        CustomerService service = new CustomerService(repository);

        // Use the service
        System.out.println(x:"Fetching customer C001:");
        service.getCustomerInfo(id:"C001");

        System.out.println(x:"\nFetching customer C003:");
        service.getCustomerInfo(id:"C003");
    }
}
```

Output

```
PS C:\Codes\DIgital Nurture\DSA\DependencyInjectionExample> javac *.java
>> java DependencyInjectionDemo
>>
Fetching customer C001:
Customer ID: C001
Customer Name: John Doe

Fetching customer C003:
Customer not found.
```