Design pattern and Principles

Exercise 1: Implementing the Singleton Pattern

Steps:

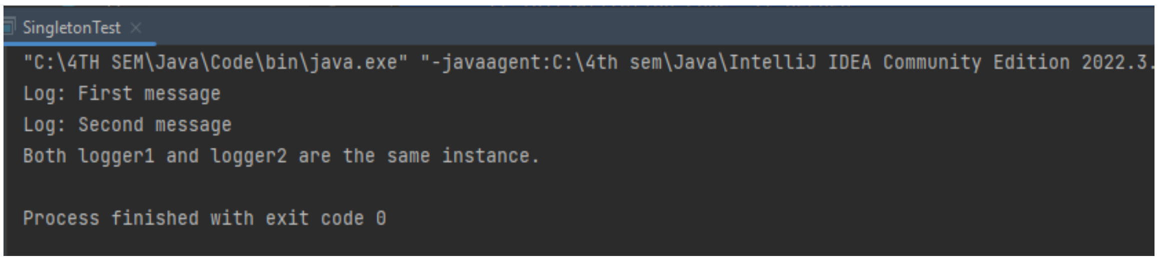
1. **Create a New Java Project:**
2. **Define a Singleton Class:**

```java
package com.example;

import com.example.singleton.Logger;

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

        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();


        logger1.log("First message");
        logger2.log("Second message");


        if (logger1 == logger2) {
            System.out.println("Both logger1 and logger2 are the same 
instance.");
        } else {
            System.out.println("logger1 and logger2 are different 
instances.");
        }
    }
}
```

3. **Implement the Singleton Pattern:**

```java
package com.example.singleton;

public class Logger {
    // Step 3: Define a private static instance of Logger
    private static Logger instance;

    // Step 4: Make the constructor private to prevent instantiation
    private Logger() {
        // Initialization code, if needed
    }

    // Step 5: Provide a public static method to get the instance of the 
Logger class
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    // A method to log messages
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

4.Test the Singleton Implementation:



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Edition 2022.3.
Log: First message
Log: Second message
Both logger1 and logger2 are the same instance.

Process finished with exit code 0
```

Exercise 2: Implementing the Factory Method Pattern

Steps:

1. **Create a New Java Project:**
2. **Define Document Classes:**

```java
package com.example.documents;

public interface Document {
    void open();
    void save();
    void close();
}
```

3. **Create Concrete Document Classes:**

```java
package com.example.documents;

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

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

    @Override
    public void close() {
        System.out.println("Closing Word document.");
    }
}
```

```java
     package com.example.documents;

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

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

    @Override
    public void close() {
        System.out.println("Closing PDF document.");
    }
}
```

```java
package com.example.documents;

public class ExcelDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Excel document.");
    }

    @Override
    public void save() {
        System.out.println("Saving Excel document.");
    }

    @Override
    public void close() {
        System.out.println("Closing Excel document.");
    }
}
```

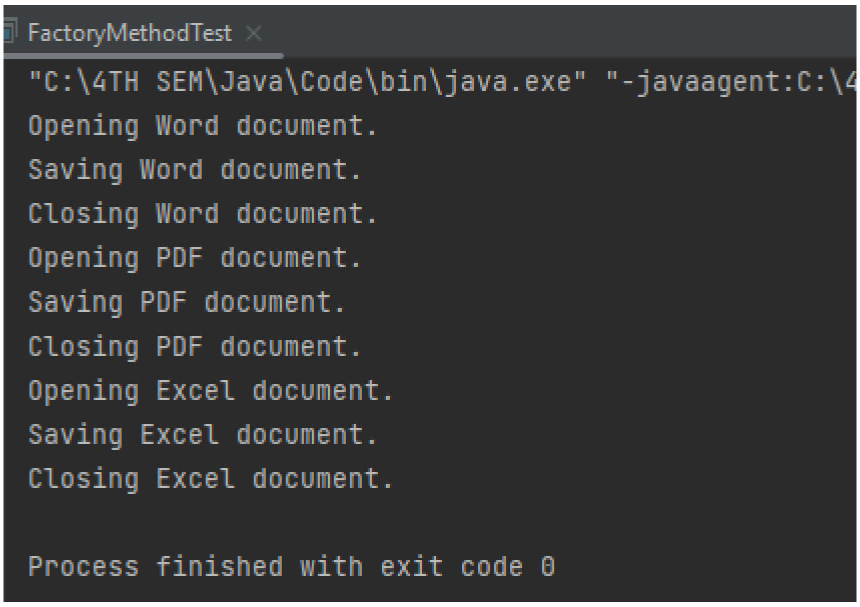
4. **Implement the Factory Method**

```java
package com.example.factory;

import com.example.documents.Document;

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

5. **Test the Factory Method Implementation:**

```java
6. package com.example.test;

   import com.example.factory.DocumentFactory;
   import com.example.factory.WordDocumentFactory;
   import com.example.factory.PdfDocumentFactory;
   import com.example.factory.ExcelDocumentFactory;
   import com.example.documents.Document;

   public class FactoryMethodTest {
       public static void main(String[] args) {
           DocumentFactory wordFactory = new WordDocumentFactory();
```

```java
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();
        wordDoc.save();
        wordDoc.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();
        pdfDoc.save();
        pdfDoc.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
        excelDoc.save();
        excelDoc.close();
    }
}
```



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4
Opening Word document.
Saving Word document.
Closing Word document.
Opening PDF document.
Saving PDF document.
Closing PDF document.
Opening Excel document.
Saving Excel document.
Closing Excel document.

Process finished with exit code 0
```

Exercise 3: Implementing the Builder Pattern

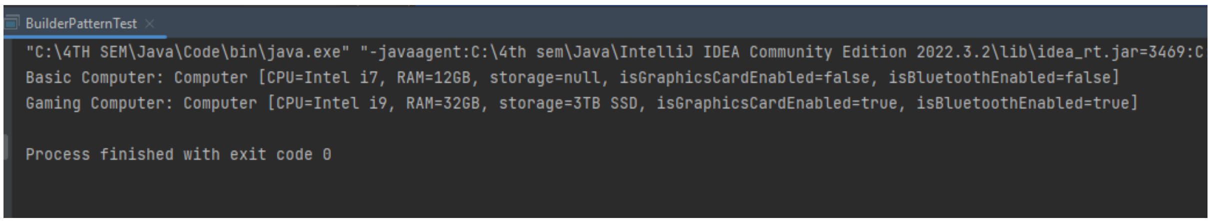
Steps:

1. Create a New Java Project:
2. **Define a Product Class:**
3. Implement the Builder Class
4. Implement the Builder Pattern:

```java
package com.example.builder;

public class Computer {
    // Attributes
    private String CPU;
    private String RAM;
    private String storage;
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    // Private constructor
    private Computer(ComputerBuilder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
        this.isBluetoothEnabled = builder.isBluetoothEnabled;
    }

    // Getters
    public String getCPU() {
        return CPU;
    }

    public String getRAM() {
        return RAM;
    }

    public String getStorage() {
        return storage;
    }

    public boolean isGraphicsCardEnabled() {
        return isGraphicsCardEnabled;
    }

    public boolean isBluetoothEnabled() {
        return isBluetoothEnabled;
    }

    // Builder class
    public static class ComputerBuilder {
        // Required attributes
        private String CPU;
        private String RAM;

        // Optional attributes
        private String storage;
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        // Constructor for required attributes
        public ComputerBuilder(String CPU, String RAM) {
            this.CPU = CPU;
            this.RAM = RAM;
        }

        // Methods for optional attributes
        public ComputerBuilder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public ComputerBuilder setGraphicsCardEnabled(boolean 
isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public ComputerBuilder setBluetoothEnabled(boolean
isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        // Build method
        public Computer build() {
            return new Computer(this);
        }
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage="
+ storage +
               ", isGraphicsCardEnabled=" + isGraphicsCardEnabled +
               ", isBluetoothEnabled=" + isBluetoothEnabled + "]";
    }
}
```

5. **Test the Builder Implementation:**

```java
package com.example.test;

import com.example.builder.Computer;

public class BuilderPatternTest {
    public static void main(String[] args) {
        Computer basicComputer = new Computer.ComputerBuilder("Intel i7",
"12GB")
                .build();

        Computer gamingComputer = new Computer.ComputerBuilder("Intel i9",
"32GB")
                .setStorage("3TB SSD")
                .setGraphicsCardEnabled(true)
                .setBluetoothEnabled(true)
                .build();

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



BuilderPatternTest

```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=3469:C
Basic Computer: Computer [CPU=Intel i7, RAM=12GB, storage=null, isGraphicsCardEnabled=false, isBluetoothEnabled=false]
Gaming Computer: Computer [CPU=Intel i9, RAM=32GB, storage=3TB SSD, isGraphicsCardEnabled=true, isBluetoothEnabled=true]

Process finished with exit code 0
```

Exercise 4: Implementing the Adapter Pattern

Steps:

1. **Create a New Java Project:**
2. Define Target Interface

```java
package com.example.payment;

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

3. **Implement Adaptee Classes:**

```java
package com.example.payment;

public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of $" + amount + "
through PayPal.");
    }
}
```

```java
package com.example.payment;

public class Stripe {
    public void makePayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through
Stripe.");
    }
}
```

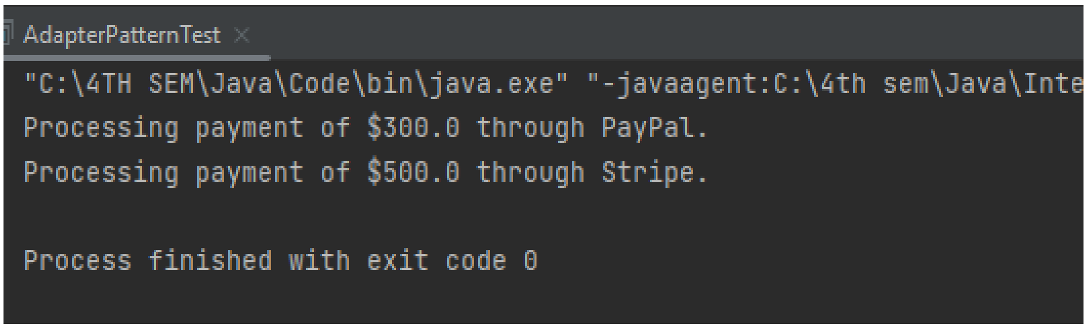
4. Implement the Adapter Class:

```java
package com.example.payment;

public class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

    public PayPalAdapter(PayPal payPal) {
        this.payPal = payPal;
    }

    @Override
    public void processPayment(double amount) {
        payPal.sendPayment(amount);
    }
}
```

```java
package com.example.payment;

public class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;

    public StripeAdapter(Stripe stripe) {
        this.stripe = stripe;
    }

    @Override
    public void processPayment(double amount) {
        stripe.makePayment(amount);
    }
}
```

5. Test the Adapter Implementation:

```
package com.example.test;

import com.example.payment.*;

public class AdapterPatternTest {
    public static void main(String[] args) {
        PaymentProcessor payPalProcessor = new PayPalAdapter(new PayPal());
        payPalProcessor.processPayment(300.0);

        PaymentProcessor stripeProcessor = new StripeAdapter(new Stripe());
        stripeProcessor.processPayment(500.0);
    }
}
```



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\Inte
Processing payment of $300.0 through PayPal.
Processing payment of $500.0 through Stripe.

Process finished with exit code 0
```

Exercise 5: Implementing the Decorator Pattern

Steps:

1.Create a New Java Project:

2 Define Component Interface:

```
package com.example.notification;

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

3.Implement Concrete Component:

```
    package com.example.notification;
```

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

4. **Implement Decorator Classes:**

```java
package com.example.notification;

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

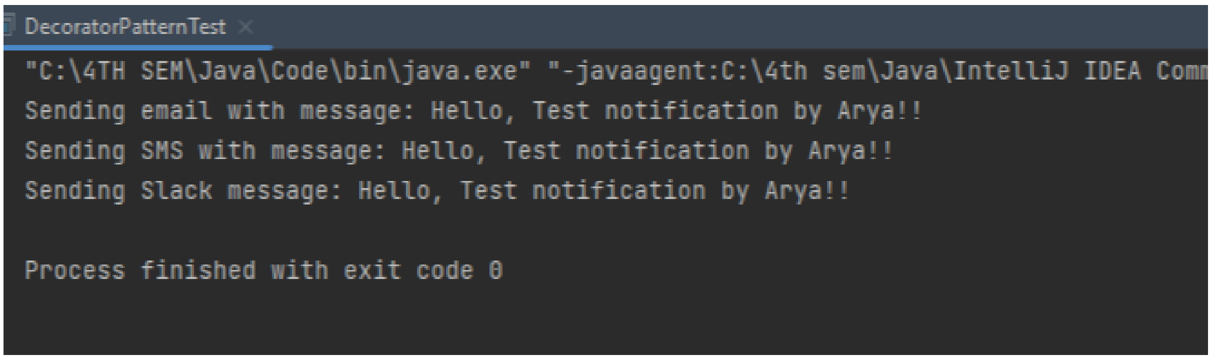
```java
package com.example.notification;

public abstract class NotifierDecorator implements Notifier {
    protected Notifier wrappedNotifier;

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

    @Override
    public void send(String message) {
        wrappedNotifier.send(message);
    }
}
```

```java
package com.example.notification;

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

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

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

```java
package com.example.notification;

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

    @Override
    public void send(String message) {
        super.send(message);
        sendSlackMessage(message);
    }

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

5. Test the Decorator Implementation:

```java
package com.example.test;

import com.example.notification.*;

public class DecoratorPatternTest {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();

        // Adding SMS Notification
        notifier = new SMSNotifierDecorator(notifier);

        // Adding Slack Notification
        notifier = new SlackNotifierDecorator(notifier);

        // Send notification
        notifier.send("Hello, Test notification by Arya!!");
    }
}
```



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Comm
Sending email with message: Hello, Test notification by Arya!!
Sending SMS with message: Hello, Test notification by Arya!!
Sending Slack message: Hello, Test notification by Arya!!

Process finished with exit code 0
```

Exercise 6: Implementing the Proxy Pattern

Steps:

1. Create a New Java Project:
2. Define Subject Interface

```java
package com.example.image;

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

3. **Implement Real Subject Class:**

```
package com.example.image;

public class RealImage implements Image {
    private String filename;

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

    private void loadImageFromDisk() {
        System.out.println("Loading image from disk: " + filename);
    }

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

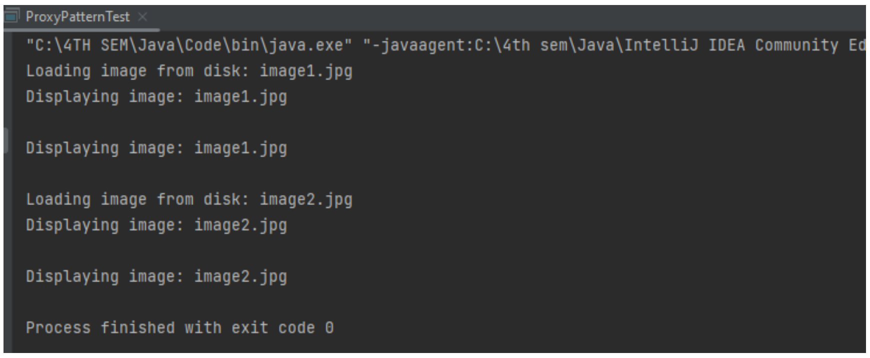
4. **Implement Proxy Class:**

```
package com.example.image;

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

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

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

5. **Test the Proxy Implementation:**

```
package com.example.test;

import com.example.image.Image;
import com.example.image.ProxyImage;

public class ProxyPatternTest {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        Image image2 = new ProxyImage("image2.jpg");

        // Image will be loaded from disk
        image1.display();
```

```java
        System.out.println("");

        // Image will not be loaded from disk
        image1.display();
        System.out.println("");

        // Image will be loaded from disk
        image2.display();
        System.out.println("");

        // Image will not be loaded from disk
        image2.display();
    }
}
```



ProxyPatternTest

```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Ed
Loading image from disk: image1.jpg
Displaying image: image1.jpg

Displaying image: image1.jpg

Loading image from disk: image2.jpg
Displaying image: image2.jpg

Displaying image: image2.jpg

Process finished with exit code 0
```

Exercise 7: Implementing the Observer Pattern

Steps:

1.Create a New Java Project:

2. Define Subject Interface:

```java
package com.example.stock;

import java.util.Observer;

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

3. Implement Concrete Subject:

```java
package com.example.stock;

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

```java
public class StockMarket implements Stock {
    private List<Observer> observers;
    private double price;

    public StockMarket() {
        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(price);
        }
    }

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

4. **Define Observer Interface:**

```java
package com.example.stock;

public interface Observer {
    void update(double price);
}
```

6. **Implement Concrete Observers:**

```java
package com.example.stock;

public class MobileApp implements Observer {
    private String name;

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

    @Override
    public void update(double price) {
        System.out.println("MobileApp " + name + " received stock price
update: " + price);
    }
}
```

```java
package com.example.stock;

public class WebApp implements Observer {
    private String name;

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

    @Override
    public void update(double price) {
        System.out.println("WebApp " + name + " received stock price
update: " + price);
    }
}
```

7. **Test the Observer Implementation:**

```java
package com.example.test;

import com.example.stock.*;

public class ObserverPatternTest {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobileApp1 = new MobileApp("App1");
        Observer mobileApp2 = new MobileApp("App2");
        Observer webApp1 = new WebApp("Web1");

        stockMarket.registerObserver((java.util.Observer) mobileApp1);
        stockMarket.registerObserver((java.util.Observer) mobileApp2);
        stockMarket.registerObserver((java.util.Observer) webApp1);

        stockMarket.setPrice(100.0);
        System.out.println();

        stockMarket.removeObserver((java.util.Observer) mobileApp1);
        stockMarket.setPrice(150.0);
    }
}
```

Exercise 8: Implementing the Strategy Pattern

Steps:

1.Create a New Java Project:

2.Define Strategy Interface:

```java
package com.example.payment;

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

3. Implement Concrete Strategies:

```
package com.example.payment;

public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cardHolderName;
    private String cvv;
    private String expiryDate;

    public CreditCardPayment(String cardNumber, String cardHolderName,
String cvv, String expiryDate) {
        this.cardNumber = cardNumber;
        this.cardHolderName = cardHolderName;
        this.cvv = cvv;
        this.expiryDate = expiryDate;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}
```

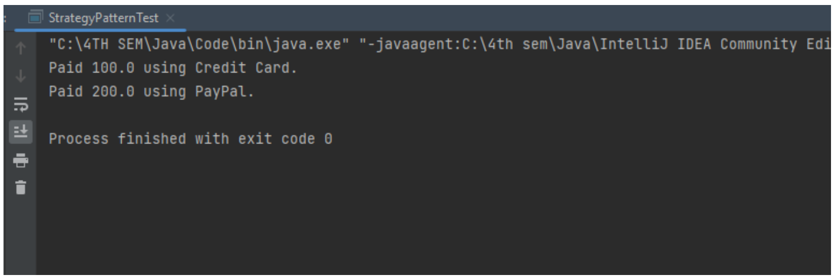
```
package com.example.payment;

public class PayPalPayment implements PaymentStrategy {
    private String email;
    private String password;

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

    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

4.Implement Context Class:

```
package com.example.payment;

public class PaymentContext {
    private PaymentStrategy paymentStrategy;

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

    public void executePayment(double amount) {
        paymentStrategy.pay(amount);
    }
}
```

6. **Test the Strategy Implementation:**

```
7. package com.example.test;

   import com.example.payment.*;

   public class StrategyPatternTest {
       public static void main(String[] args) {
           PaymentContext context = new PaymentContext();

           // Pay using Credit Card
           PaymentStrategy creditCardPayment = new
   CreditCardPayment("1234567890123456", "John Doe", "123", "12/23");
           context.setPaymentStrategy(creditCardPayment);
           context.executePayment(100.0);

           // Pay using PayPal
           PaymentStrategy payPalPayment = new
   PayPalPayment("johndoe@example.com", "password123");
           context.setPaymentStrategy(payPalPayment);
           context.executePayment(200.0);
       }
   }
```



StrategyPatternTest

```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Edi
Paid 100.0 using Credit Card.
Paid 200.0 using PayPal.

Process finished with exit code 0
```

Exercise 9: Implementing the Command Pattern

Steps:

1. **Create a New Java Project:**
2. **Define Command Interface:**

```
package com.example.command;

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

3. **Implement Concrete Commands:**

```
package com.example.command;

public class LightOnCommand implements Command {
    private Light light;

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

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

```
package com.example.command;

public class LightOffCommand implements Command {
    private Light light;

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

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

4. **Implement Invoker Class:**

```
package com.example.command;

public class RemoteControl {
    private Command command;

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

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

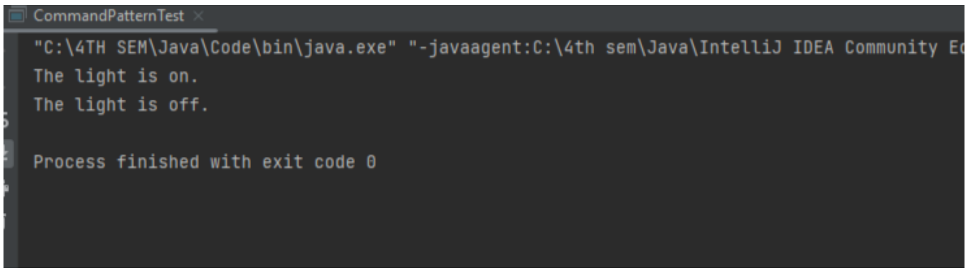
5. **Implement Receiver Class:**

```
package com.example.command;

public class Light {
    public void turnOn() {
        System.out.println("The light is on.");
    }

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

6. **Test the Command Implementation:**

```
package com.example.test;

import com.example.command.*;
```

```java
public class CommandPatternTest {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        // Turn on the light
        remote.setCommand(lightOn);
        remote.pressButton();

        // Turn off the light
        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Ec
The light is on.
The light is off.

Process finished with exit code 0
```

Exercise 10: Implementing the MVC Pattern

Steps:

1. **Create a New Java Project**
2. **Define Model Class**

```java
package com.example.model;

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;
    }

    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;
    }
}
```

3. **Define View Class**

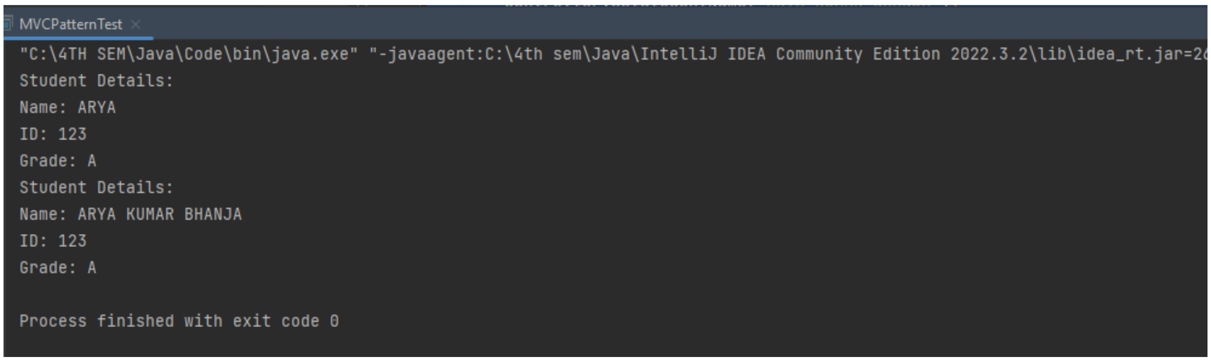
```
package com.example.view;

public class StudentView {
    public void displayStudentDetails(String studentName, String
studentId, String studentGrade) {
        System.out.println("Student Details:");
        System.out.println("Name: " + studentName);
        System.out.println("ID: " + studentId);
        System.out.println("Grade: " + studentGrade);
    }
}
```

4. **Define Controller Class**

```
package com.example.controller;

import com.example.model.Student;
import com.example.view.StudentView;

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 String getStudentName() {
        return model.getName();
    }

    public void setStudentId(String id) {
        model.setId(id);
    }

    public String getStudentId() {
        return model.getId();
    }

    public void setStudentGrade(String grade) {
        model.setGrade(grade);
    }

    public String getStudentGrade() {
        return model.getGrade();
    }

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

5. **Test the MVC Implementation:**

```
package com.example.test;

import com.example.model.Student;
import com.example.view.StudentView;
import com.example.controller.StudentController;

public class MVCPatternTest {
    public static void main(String[] args) {
        // Create a student
        Student student = new Student("ARYA", "123", "A");

        // Create a view to display student details
        StudentView view = new StudentView();

        // Create a controller to handle the communication between
model and view
        StudentController controller = new StudentController(student,
view);

        // Display the initial details
        controller.updateView();

        // Update student details
        controller.setStudentName("ARYA KUMAR BHANJA");
        controller.setStudentGrade("A");

        // Display the updated details
        controller.updateView();
    }
}
```



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=2
Student Details:
Name: ARYA
ID: 123
Grade: A
Student Details:
Name: ARYA KUMAR BHANJA
ID: 123
Grade: A

Process finished with exit code 0
```

Exercise 11: Implementing Dependency Injection

Steps:

1. **Create a New Java Project**

2. **Define Repository Interface**

```java
package com.example.repository;

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

3. **Implement Concrete Repository**

```java
package com.example.repository;

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

public class CustomerRepositoryImpl implements CustomerRepository {
    private Map<String, String> customerData;

    public CustomerRepositoryImpl() {
        customerData = new HashMap<>();
        customerData.put("1", "ARYA KUMAR BHANJA");
        customerData.put("2", "SOFTWARE DEVELOPER");
    }

    @Override
    public String findCustomerById(String id) {
        return customerData.get(id);
    }
}
```

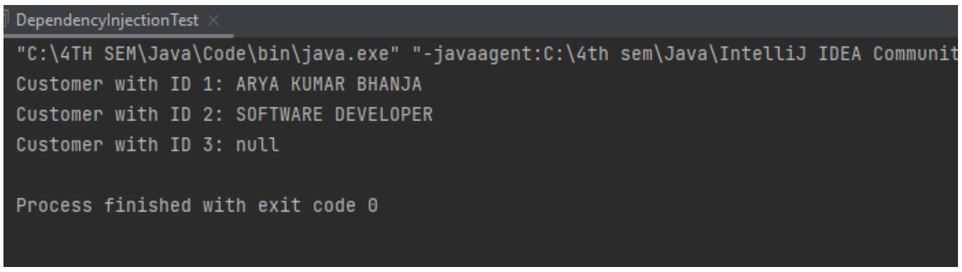
4. **Define Service Class**

```java
package com.example.service;

import com.example.repository.CustomerRepository;

public class CustomerService {
    private final CustomerRepository customerRepository;

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

    public String getCustomerById(String id) {
        return customerRepository.findCustomerById(id);
    }
}
```

5. Implement Dependency Injection

```java
package com.example.test;

import com.example.repository.CustomerRepository;
import com.example.repository.CustomerRepositoryImpl;
import com.example.service.CustomerService;

public class DependencyInjectionTest {
    public static void main(String[] args) {
        // Create the repository
        CustomerRepository repository = new CustomerRepositoryImpl();

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

        // Use the service to find a customer
        String customer = service.getCustomerById("1");
        System.out.println("Customer with ID 1: " + customer);

        customer = service.getCustomerById("2");
        System.out.println("Customer with ID 2: " + customer);

        customer = service.getCustomerById("3");
        System.out.println("Customer with ID 3: " + customer);  // This
should return null
    }
}
```

6. Test the Dependency Injection Implementation



```
"C:\4TH SEM\Java\Code\bin\java.exe" "-javaagent:C:\4th sem\Java\IntelliJ IDEA Communit
Customer with ID 1: ARYA KUMAR BHANJA
Customer with ID 2: SOFTWARE DEVELOPER
Customer with ID 3: null

Process finished with exit code 0
```