DESIGN PATTERNS AND PRINCIPLES

**Exercise 1: Implementing the Singleton Pattern**

**SOLUTION**

**Logger.java**

package SingletonPattern;

public class Logger {

private static Logger instance;

private Logger() {

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("Log Message: " + message);

}

}

**TestSingleton.java**

package SingletonPattern;

import SingletonPattern.Logger;

public class TestSingleton {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("This is the first log message.");

Logger logger2 = Logger.getInstance();

Logger logger3 = Logger.getInstance();

logger3.log("This is the third log message.");

System.out.println("\nVerifying instances:");

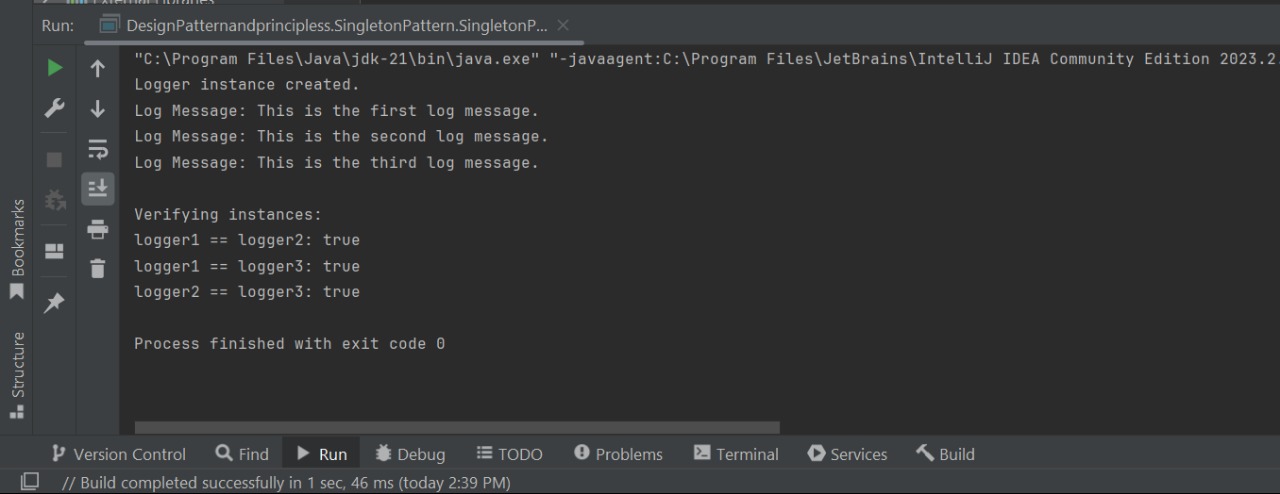
System.out.println("logger1 == logger2: " + (logger1 == logger2));

System.out.println("logger1 == logger3: " + (logger1 == logger3));

System.out.println("logger2 == logger3: " + (logger2 == logger3));

}

}

**Output:**

**Exercise 2: Implementing the Factory Method Pattern**

**SOLUTION**

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

public void open() {

System.out.println("Opening a Word document.");

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

public void open() {

System.out.println("Opening a PDF document.");

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

public void open() {

System.out.println("Opening an Excel document.");

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

    }

}

**ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

    }

}

**TestFactoryPattern.java**

public class TestFactoryPattern {

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();

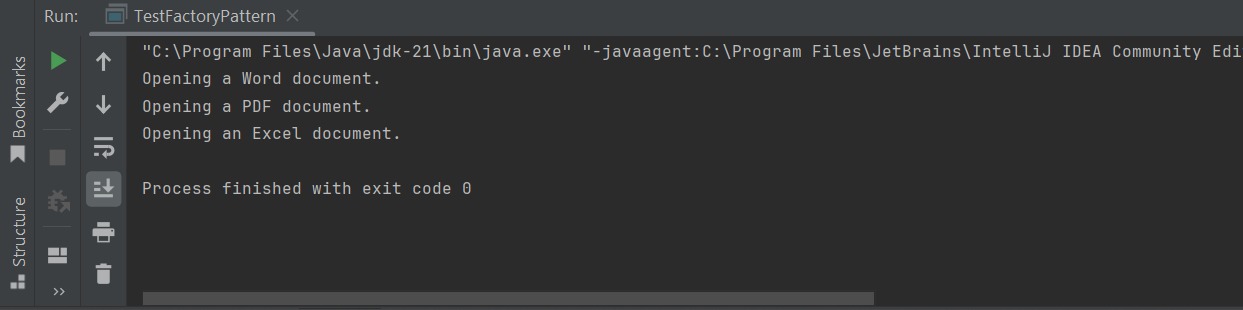
DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**

**Exercise 3: Implementing the Builder Pattern**

**SOLUTION**

**Computer.java**

Package BuilderPatternExample;

public class Computer {

private String CPU;

private String RAM

private String storage;

private String graphicsCard;

private String motherboard;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.motherboard = builder.motherboard;

}

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +

", GraphicsCard=" + graphicsCard + ", Motherboard=" + motherboard + "]";

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private String motherboard;

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 setMotherboard(String motherboard) {

this.motherboard = motherboard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

**Main.java**

package BuilderPatternExample;

public class Main {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i3", "8GB").build();

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.setGraphicsCard("NVIDIA RTX 4080")

.setStorage("1TB SSD")

.setMotherboard("ASUS ROG Maximus")

.build();

Computer workstation = new Computer.Builder("AMD Ryzen 9", "64GB")

.setStorage("2TB SSD")

.setGraphicsCard("NVIDIA Quadro RTX 5000")

.build();

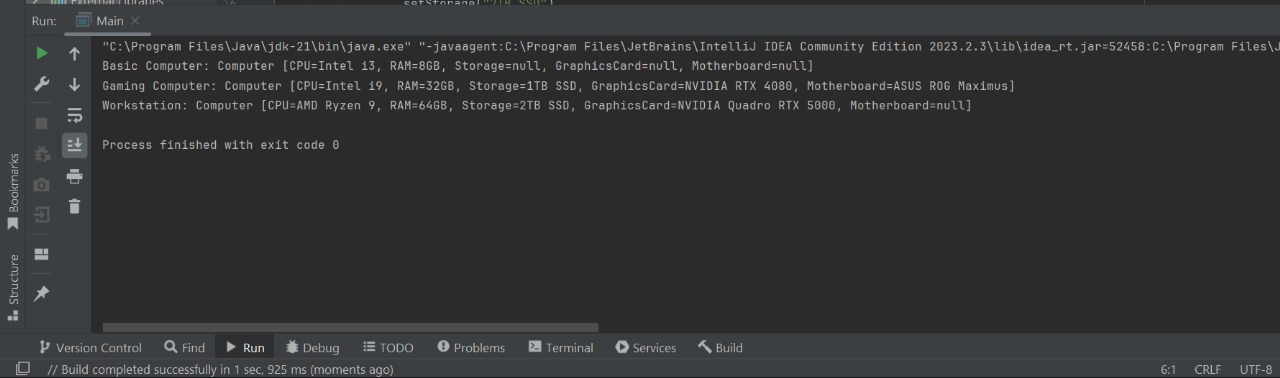
System.out.println("Basic Computer: " + basicComputer);

System.out.println("Gaming Computer: " + gamingComputer);

System.out.println("Workstation: " + workstation);

}

}

***Output:***

**Exercise 4: Implementing the Adapter Pattern**

**SOLUTION**

***PaymentProcessor.java***

package AdapterPatternExample;

public interface PaymentProcessor {

void processPayment(double amount);

}

**PaypalGateway.java**

public class PaypalGateway {

public void sendPayment(double amountInUSD) {

System.out.println("Processing payment through PayPal: $" + amountInUSD);

    }

}

**StripeGateway.java**

public class StripeGateway {

public void makePayment(double amountInDollars) {

System.out.println("Processing payment through Stripe: $" + amountInDollars);

    }

}

**PaypalAdapter.java**

public class PaypalAdapter implements PaymentProcessor {

private PaypalGateway paypal;

public PaypalAdapter(PaypalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

    }

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makePayment(amount);

    }}

**Main.java**

public class Main {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PaypalAdapter(new PaypalGateway());

paypalProcessor.processPayment(100.0);

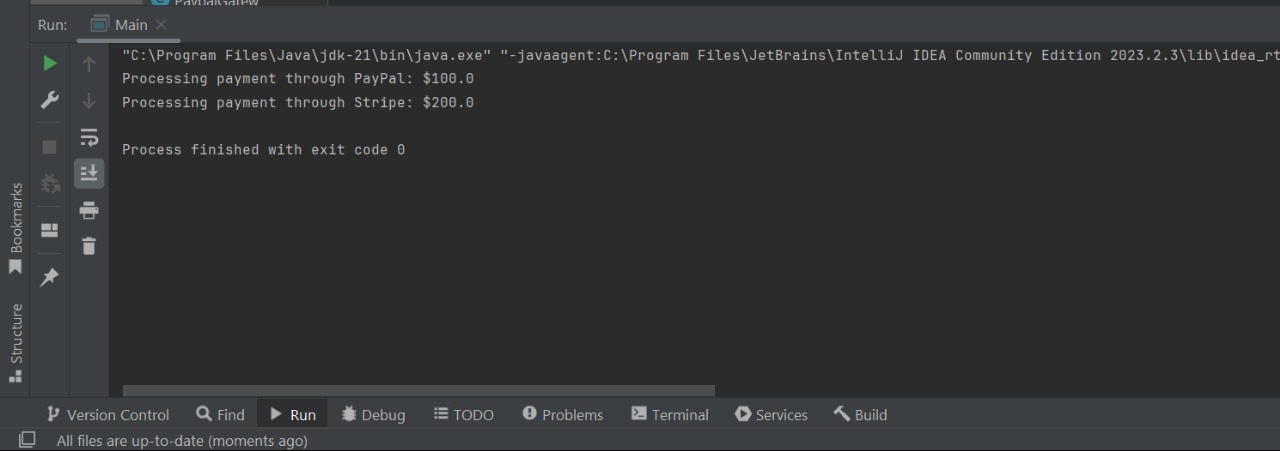
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(200.0);

    }

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**SOLUTION**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

System.out.println("Sending EMAIL: " + message);

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

@Override

public void send(String message) {

wrappee.send(message);

}

}

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: " + message);

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); sendSlack(message);

}

private void sendSlack(String message) {

System.out.println("Sending SLACK message: " + message);

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

Notifier emailOnly = new EmailNotifier();

emailOnly.send("System maintenance at 10 PM");

System.out.println("\n---");

Notifier emailSMS = new SMSNotifierDecorator(new EmailNotifier());

emailSMS.send("Your password has been changed");

System.out.println("\n---");

Notifier allChannels = new SlackNotifierDecorator(

new SMSNotifierDecorator(

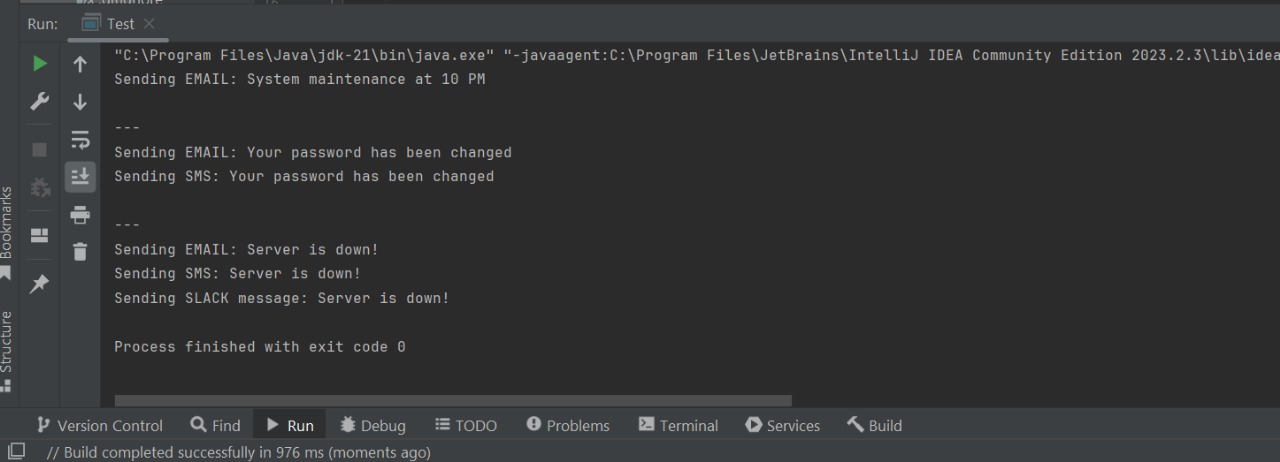
new EmailNotifier()));

allChannels.send("Server is down!");

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**SOLUTION**

Package ProxyPattern**;**

**Image.java**

public interface Image {

void display();

}

**RealImage.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);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public void display() {

System.out.println("Displaying: " + filename);

}

}

**ProxyImage.java**

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);

} else {

System.out.println("Using cached image: " + filename);

}

realImage.display();

    }

}

**Testimage.java**

public class Testimage {

public static void main(String[] args) {

Image image1 = new ProxyImage("nature.jpg");

Image image2 = new ProxyImage("cityscape.jpg");

System.out.println("First call (image1):");

image1.display();

System.out.println("\nSecond call (image1):");

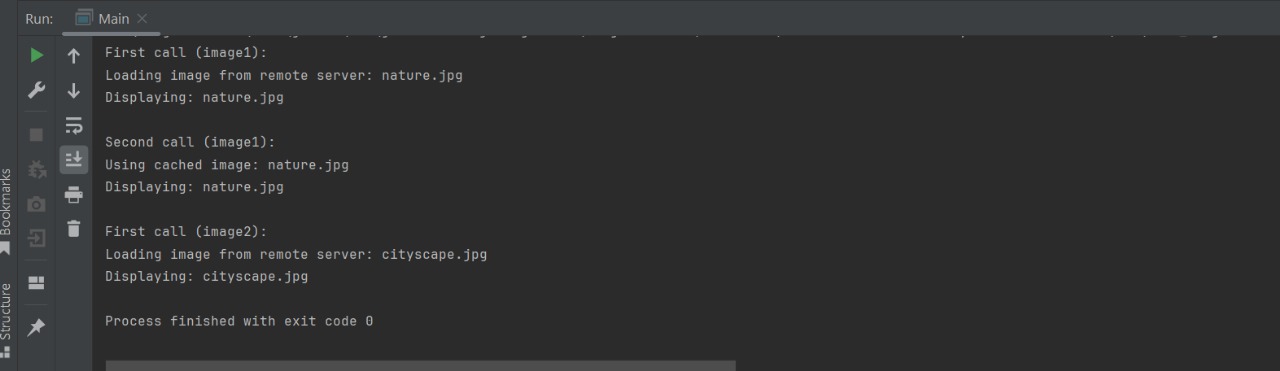
image1.display();

System.out.println("\nFirst call (image2):");

image2.display();

    }

}

**Output:**

**Exercise 7: Implementing the Observer Pattern**

**SOLUTION**

Package ObserverPatternExample;

**Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**Observer.java**

public interface Observer {

void update(String stockName, double price);

}

**StockMarket.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();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}}

**MobileApp.java**

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println("MobileApp " + name + ": " + stockName + " price updated to " + price);

}}

**WebApp.java**

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println("WebApp " + name + ": " + stockName + " price updated to " + price);

}}

**ObserverPatternTest.java**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileApp1 = new MobileApp("Investor A");

Observer webApp1 = new WebApp("Investor B");

market.registerObserver(mobileApp1);

market.registerObserver(webApp1);

market.setStockPrice("AAPL", 178.34);

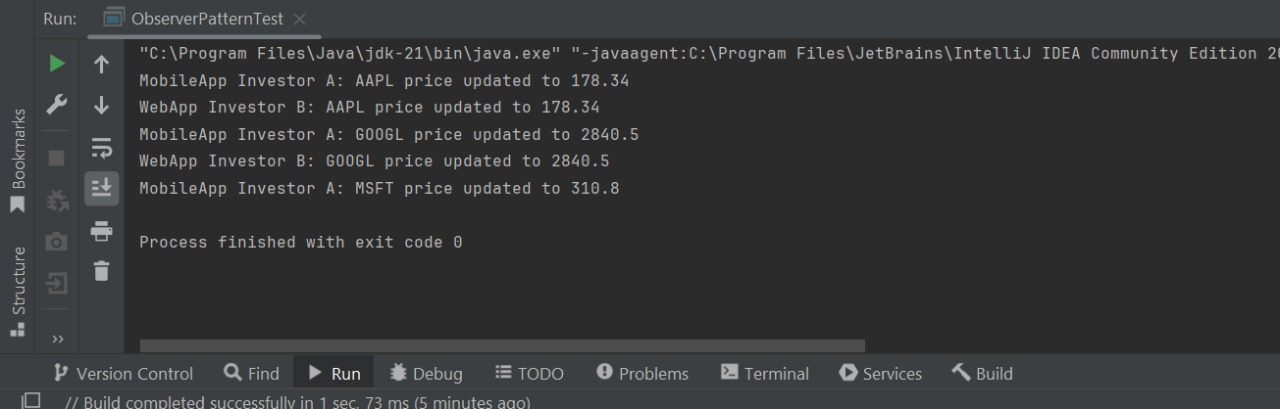
market.setStockPrice("GOOGL", 2840.50);

market.removeObserver(webApp1);

market.setStockPrice("MSFT", 310.80);

}

}

**Output:**

**Exercise 8: Implementing the Strategy Pattern**

**SOLUTION**

Package StrategyPatternExample;

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder; }

public void pay(double amount) {

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

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

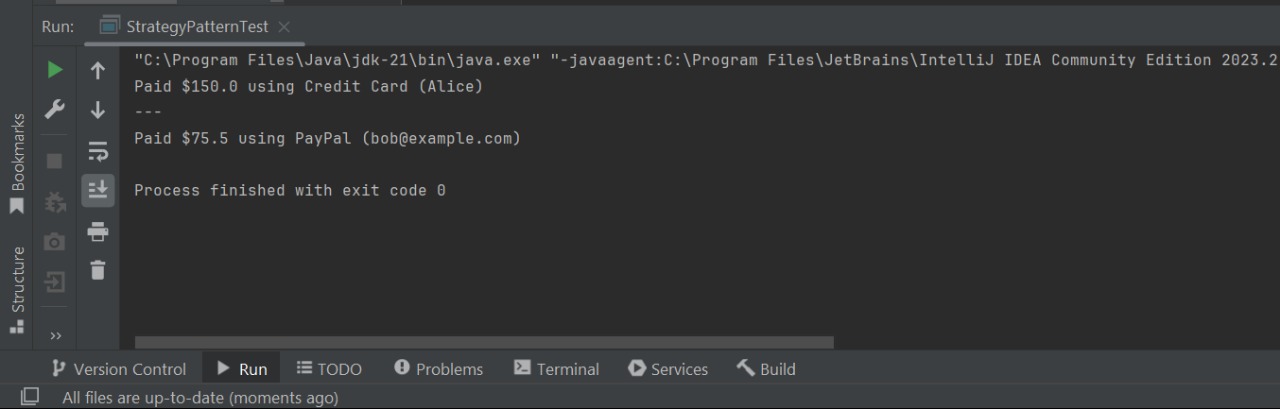
this.email = email;}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal (" + email + ")");

}}

***Output:***



**Exercise 9: Implementing the Command Pattern**

**SOLUTION**

Package CommandPatternExample;

**Command.java**

public interface Command {

void execute();

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**Light.java**

public class Light {

public void turnOn() {

System.out.println("The light is ON"); }

public void turnOff() {

System.out.println("The light is OFF");

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

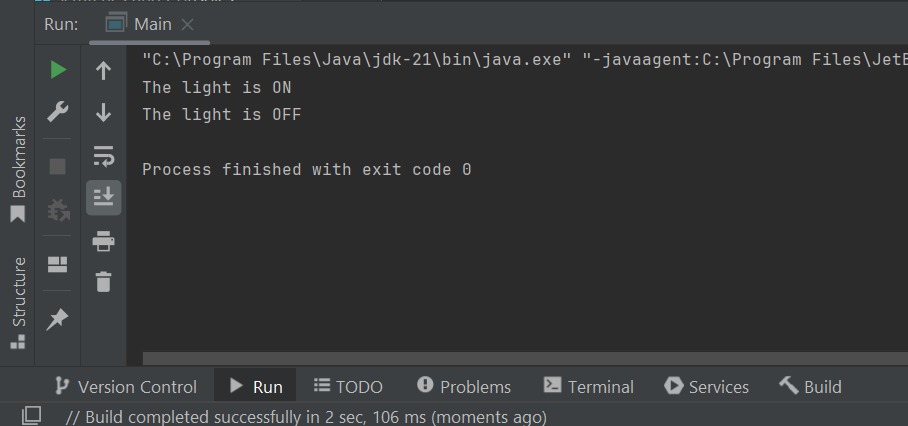
remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**

**Exercise 10: Implementing the MVC Pattern**

**SOLUTION**

**Student.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;

}

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(String name, String id, String grade) {

System.out.println("---- Student Details ----");

System.out.println("Name : " + name);

System.out.println("ID : " + id);

System.out.println("Grade : " + grade);

System.out.println("--------------------------");

}}

**StudentController.java**

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() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade()); }}

**Main.java**

public class Main {

public static void main(String[] args) {

Student student = new Student("Alice", "S101", "A");

StudentView view = new StudentView();

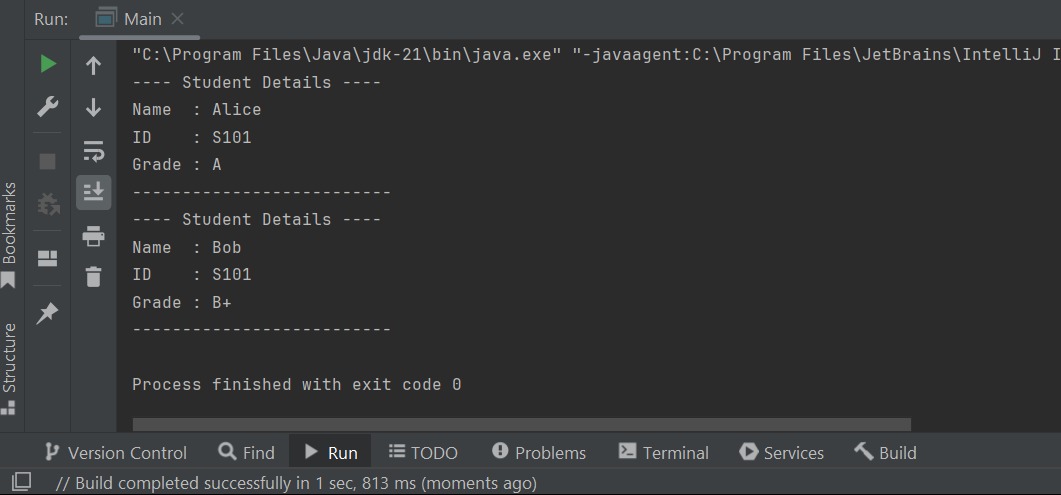
StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Bob");

controller.setStudentGrade("B+");

controller.updateView(); }}

**Output:**

**Exercise 11: Implementing Dependency Injection**

**SOLUTION**

**Customer.java**

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

public String toString() {

return "Customer ID: " + id + ", Name: " + name;

}

}

**CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

private Map<String, Customer> customerMap = new HashMap<>();

public CustomerRepositoryImpl() {

customerMap.put("C001", new Customer("C001", "Alice"));

customerMap.put("C002", new Customer("C002", "Bob"));

}

public Customer findCustomerById(String id) {

return customerMap.get(id);

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

Customer customer = repository.findCustomerById(id);

if (customer != null) {

System.out.println(customer);

} else {

System.out.println("Customer not found.");

}

}

***}***

**Main.java**

public class Main {

public static void main(String[] args) {

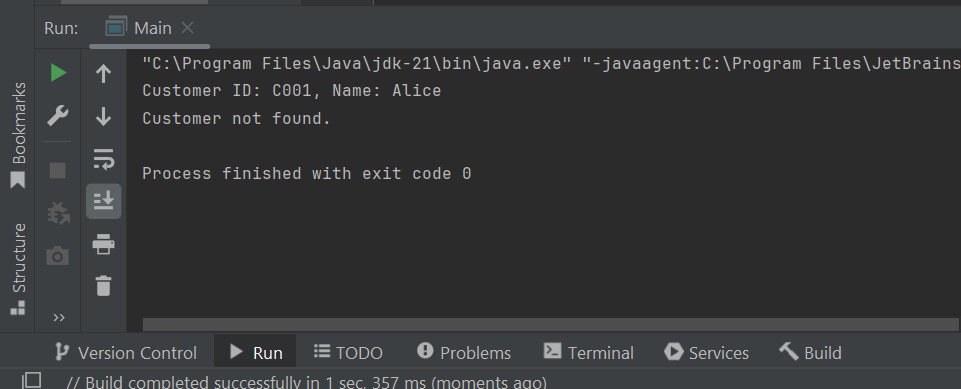
CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomerDetails("C001");

service.getCustomerDetails("C003");

}}

***Output:***