**WEEK 1 DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

class Logger {

    private static Logger instance = new Logger();

    private Logger() {

        System.out.println("Logger instance created");

    }

    public static Logger getInstance() {

        return instance;

    }

    public void log(String message) {

        System.out.println("[LOG] " + message);

    }

}

public class SingletonPatternExample {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        Logger logger3 = Logger.getInstance();

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

        logger2.log("This is the second log message.");

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

        if (logger1 == logger2 && logger2 == logger3) {

            System.out.println("Only one instance of Logger exists.");

        } else {

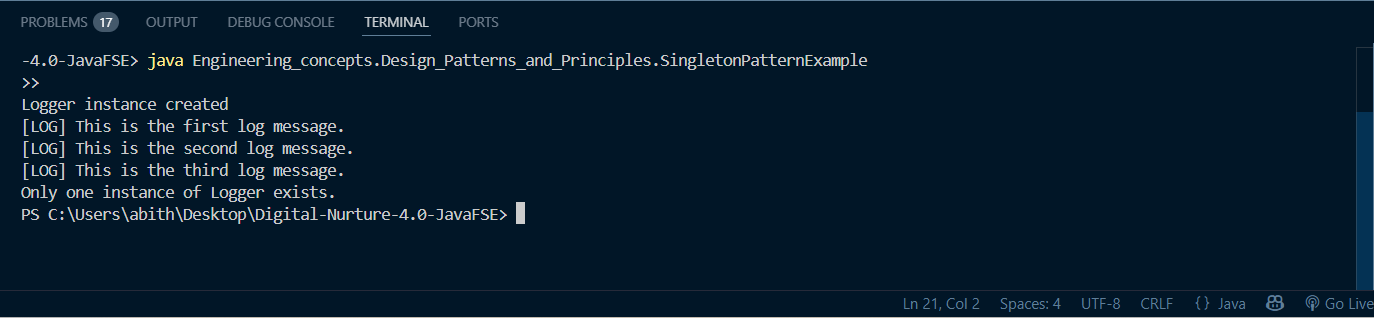
            System.out.println("Multiple instances detected! Singleton failed.");

      }

    }

}

**OUTPUT:**



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

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

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 {

    public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

public class FactoryMethodPatternExample {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

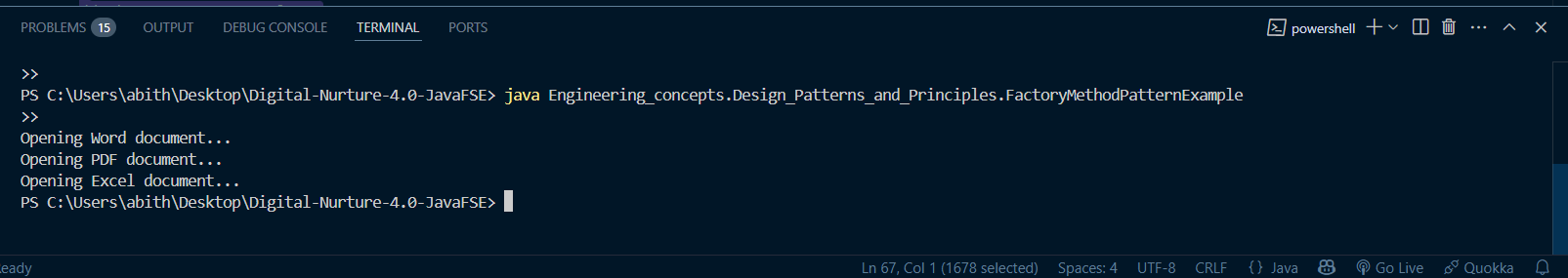
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

OUTPUT:



**Exercise 3: Implementing the Builder Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

class Computer {

    private String CPU;

    private String RAM;

    private String Storage;

    private String GraphicsCard;

    private boolean Bluetooth;

    private boolean WiFi;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.Storage = builder.Storage;

        this.GraphicsCard = builder.GraphicsCard;

        this.Bluetooth = builder.Bluetooth;

        this.WiFi = builder.WiFi;

    }

    public void displayConfiguration() {

        System.out.println("Computer Configuration:");

        System.out.println("CPU: " + CPU);

        System.out.println("RAM: " + RAM);

        System.out.println("Storage: " + Storage);

        System.out.println("Graphics Card: " + GraphicsCard);

        System.out.println("Bluetooth: " + (Bluetooth ? "Yes" : "No"));

        System.out.println("WiFi: " + (WiFi ? "Yes" : "No"));

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String Storage;

        private String GraphicsCard;

        private boolean Bluetooth;

        private boolean WiFi;

        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 Builder setGraphicsCard(String GraphicsCard) {

            this.GraphicsCard = GraphicsCard;

            return this;

        }

        public Builder setBluetooth(boolean Bluetooth) {

            this.Bluetooth = Bluetooth;

            return this;

        }

        public Builder setWiFi(boolean WiFi) {

            this.WiFi = WiFi;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

public class BuilderPatternExample {

    public static void main(String[] args) {

        Computer highEndComputer = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("2TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .setBluetooth(true)

                .setWiFi(true)

                .build();

highEndComputer.displayConfiguration();

  System.out.println();

        Computer budgetComputer = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("8GB")

                .setStorage("512GB SSD")

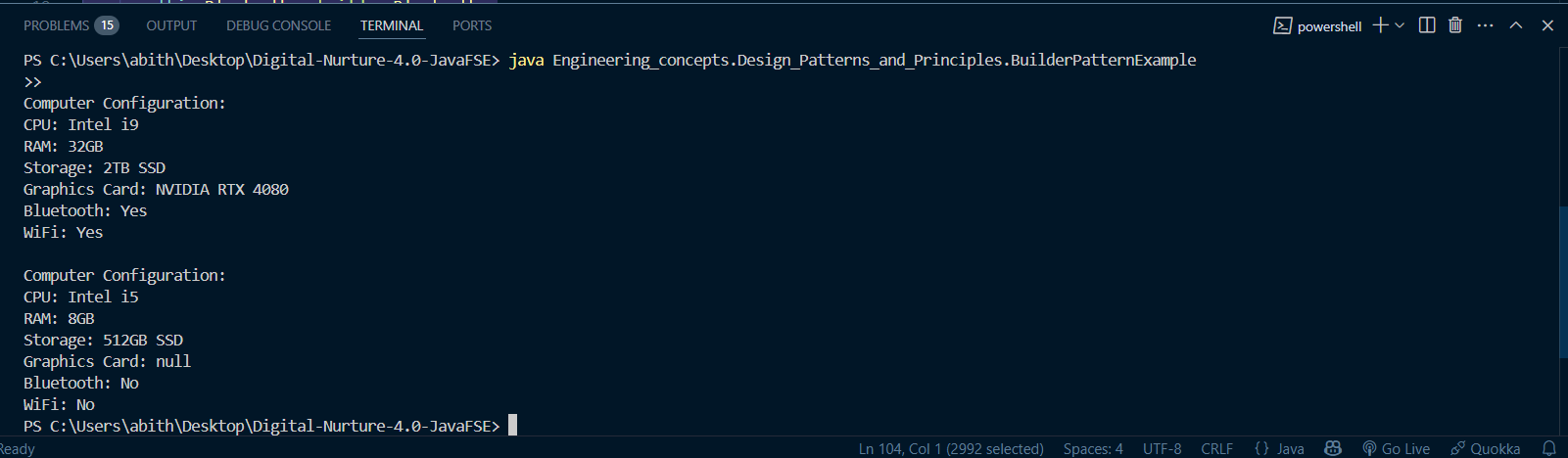
                .build();

        budgetComputer.displayConfiguration();

    }

}

OUTPUT:



**Exercise 4: Implementing the Adapter Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

interface PaymentProcessor {

    void processPayment(double amount);

}

class PayPalGateway {

    public void sendPayment(double amount) {

        System.out.println("Processing payment of $" + amount + " via PayPal");

    }

}

class StripeGateway {

    public void makePayment(double amount) {

        System.out.println("Processing payment of $" + amount + " via Stripe");

    }

}

class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway) {

        this.payPalGateway = payPalGateway;

    }

    public void processPayment(double amount) {

        payPalGateway.sendPayment(amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway stripeGateway) {

        this.stripeGateway = stripeGateway;

    }

    public void processPayment(double amount) {

        stripeGateway.makePayment(amount);

    }

}

public class AdapterPatternExample {

    public static void main(String[] args) {

        PaymentProcessor payPalPayment = new PayPalAdapter(new PayPalGateway());

        payPalPayment.processPayment(100.0);

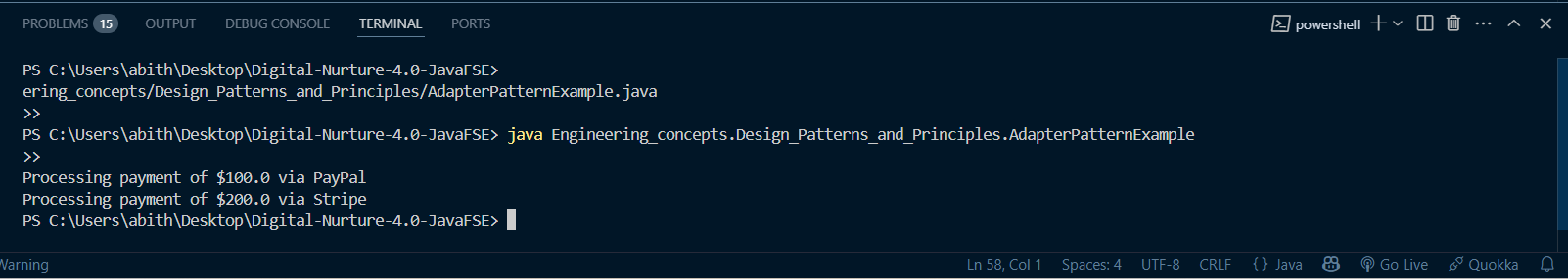
        PaymentProcessor stripePayment = new StripeAdapter(new StripeGateway());

        stripePayment.processPayment(200.0);

    }

}

OUTPUT:



**Exercise 5: Implementing the Decorator Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}

class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

public class DecoratorPatternExample {

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier();

        Notifier smsNotifier = new SMSNotifierDecorator(notifier);

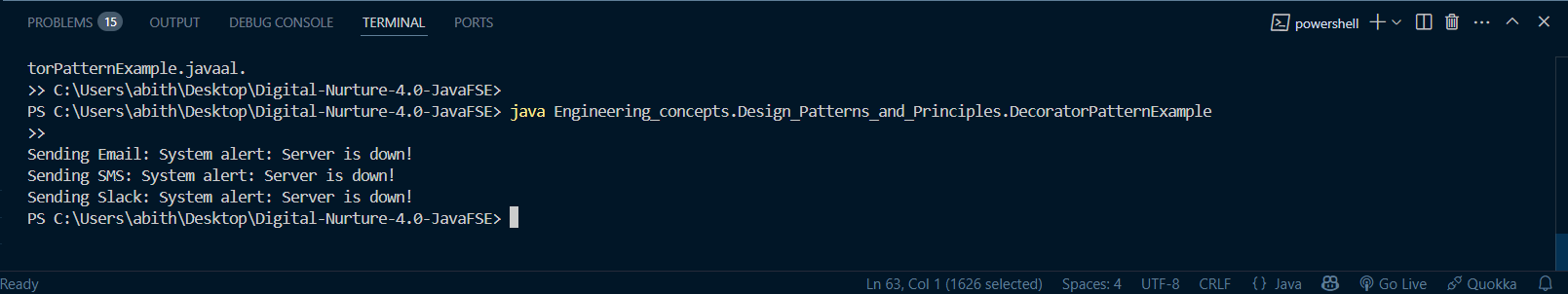
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("System alert: Server is down!");

    }

}

OUTPUT:

****

**Exercise 6: Implementing the Proxy Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

interface Image {

    void display();

}

class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromDisk();

    }

    private void loadFromDisk() {

        System.out.println("Loading " + filename);

    }

    public void display() {

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

    }

}

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();

    }

}

public class ProxyPatternExample {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("photo1.jpg");

        Image image2 = new ProxyImage("photo2.jpg");

        image1.display();

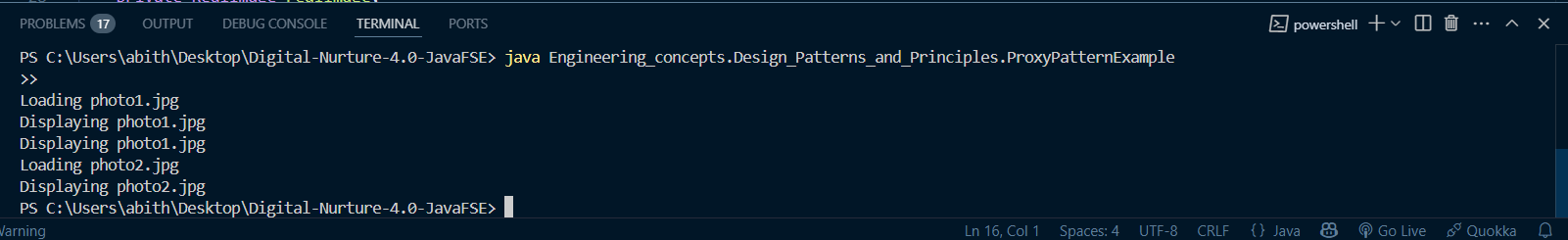
        image1.display();

        image2.display();

    }

}

OUTPUT:



**Exercise 7: Implementing the Observer Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

import java.util.\*;

interface Stock {

    void register(Observer o);

    void deregister(Observer o);

    void notifyObservers();

}

interface Observer {

    void update(float price);

}

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);

        }

    }

}

class MobileApp implements Observer {

    public void update(float price) {

        System.out.println("MobileApp: Stock price updated to $" + price);

    }

}

class WebApp implements Observer {

    public void update(float price) {

        System.out.println("WebApp: Stock price updated to $" + price);

    }

}

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobile = new MobileApp();

        Observer web = new WebApp();

        stockMarket.register(mobile);

        stockMarket.register(web);

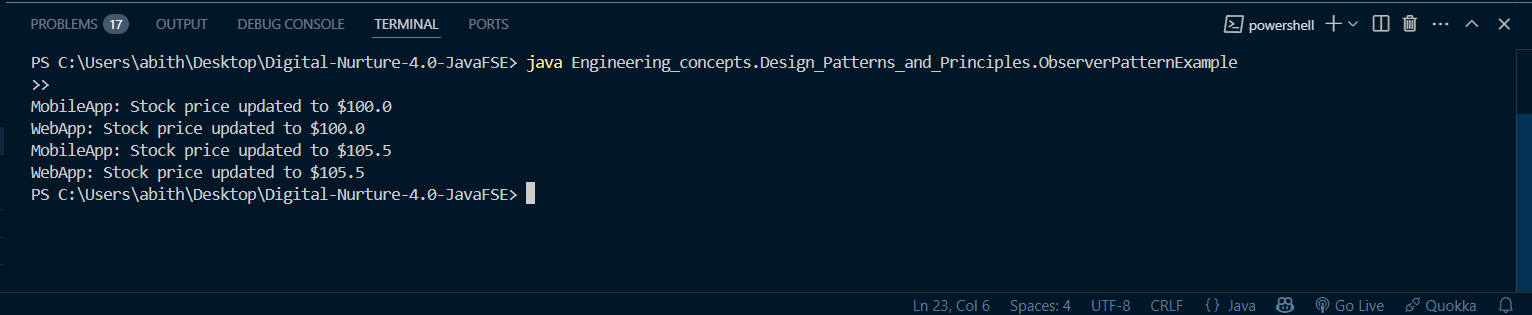
        stockMarket.setPrice(100.0f);

        stockMarket.setPrice(105.5f);

    }

}

OUTPUT:



**Exercise 8: Implementing the Strategy Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

class PayPalPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void payAmount(double amount) {

        strategy.pay(amount);

    }

}

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment());

        context.payAmount(500);

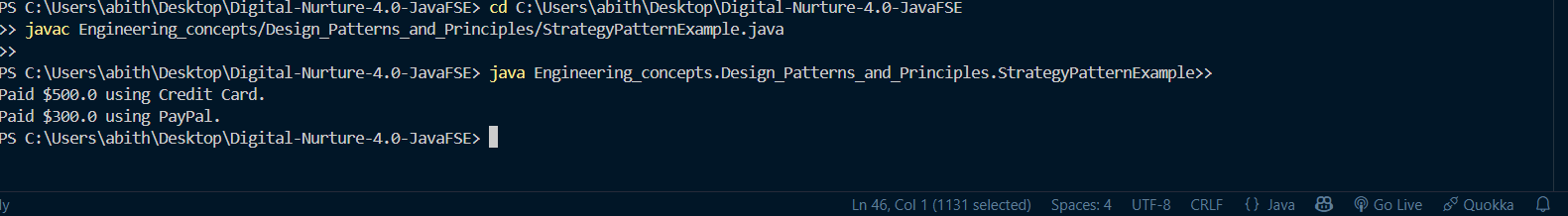
        context.setStrategy(new PayPalPayment());

        context.payAmount(300);

    }

}

OUTPUT:



**Exercise 9: Implementing the Command Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

interface Command {

    void execute();

}

class Light {

    public void turnOn() {

        System.out.println("Light is ON");

    }

    public void turnOff() {

        System.out.println("Light is OFF");

    }

}

class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOn();

    }

}

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOff();

    }

}

class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        command.execute();

    }

}

public class CommandPatternExample {

    public static void main(String[] args) {

        Light light = new Light();

        RemoteControl remote = new RemoteControl();

        remote.setCommand(new LightOnCommand(light));

        remote.pressButton();

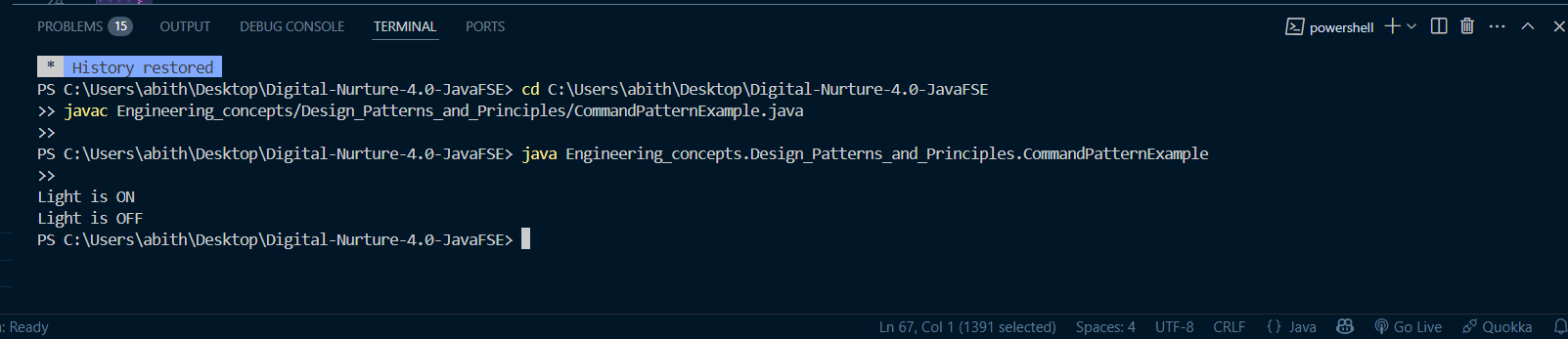
        remote.setCommand(new LightOffCommand(light));

        remote.pressButton();

    }

}

OUTPUT:



**Exercise 10: Implementing the MVC Pattern**

package Engineering\_concepts.Design\_Patterns\_and\_Principles;

class Student {

    private String id;

    private String name;

    private String grade;

    public Student(String id, String name, String grade) {

        this.id = id;

        this.name = name;

        this.grade = grade;

    }

    public String getId() { return id; }

    public String getName() { return name; }

    public String getGrade() { return grade; }

    public void setName(String name) { this.name = name; }

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

}

class StudentView {

    public void displayStudentDetails(String id, String name, String grade) {

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

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

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

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

    }

}

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.getId(), student.getName(), student.getGrade());

    }

    public void setStudentName(String name) {

        student.setName(name);

    }

    public void setStudentGrade(String grade) {

        student.setGrade(grade);

    }

}

public class MVCPatternExample {

    public static void main(String[] args) {

        Student student = new Student("101", "John", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Michael");

        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

OUTPUT:

