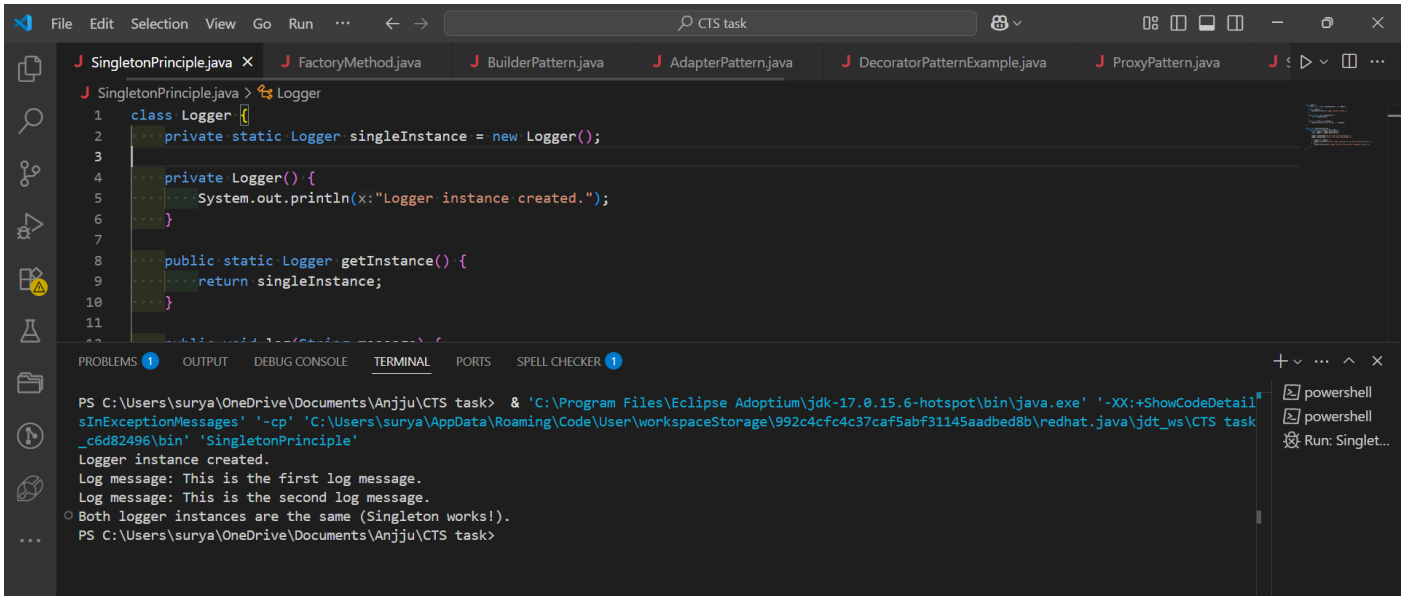
DESIGN PRINCIPLES AND DESIGN PATTERNS EXERCISES

Exercise 1: Implementing the Singleton Pattern

```
class Logger {
    private static Logger singleInstance = new Logger();

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

    public static Logger getInstance() {
        return singleInstance;
    }

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

public class SingletonPrinciple {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("This is the first log message.");
        logger2.log("This is the second log message.");

        if (logger1 == logger2) {
            System.out.println("Both logger instances are the same (Singleton works!).");
        } else {
            System.out.println("Logger instances are different (Singleton failed!).");
```

```
    }

  }

}
```

OUTPUT:

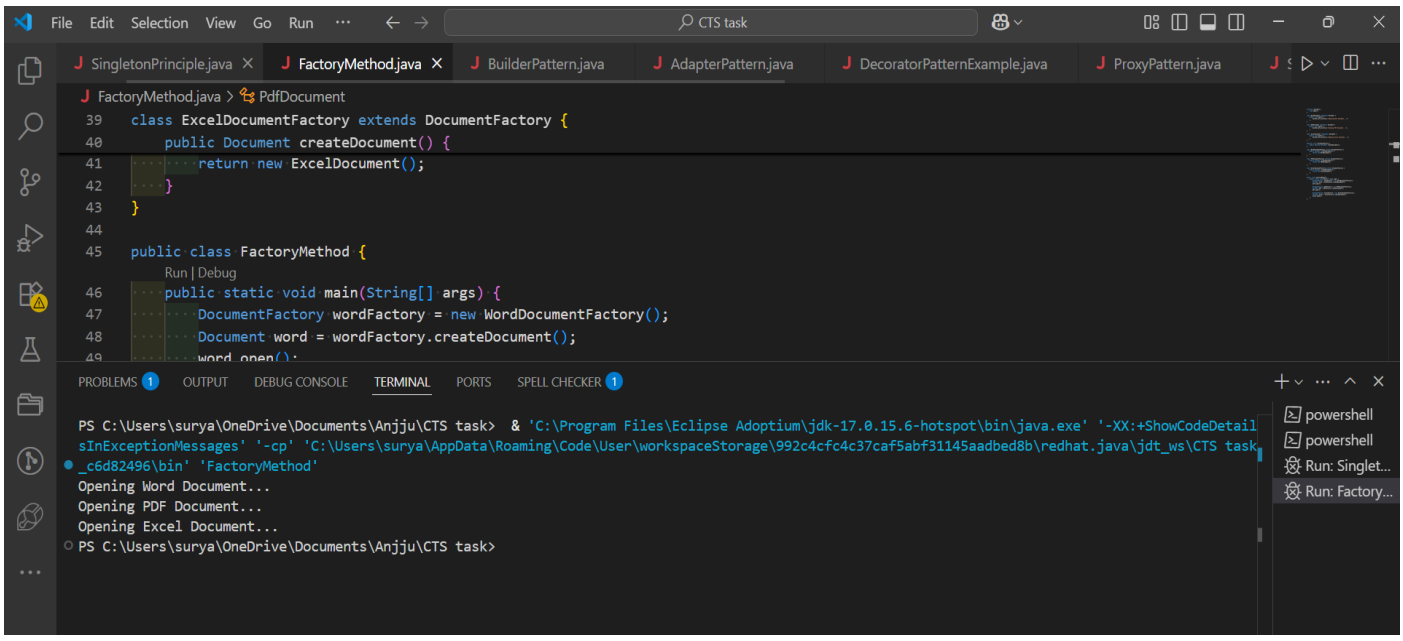


Exercise 2: Implementing the Factory Method Pattern

```
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();
}
```

```java
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 FactoryMethod {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document word = wordFactory.createDocument();
        word.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdf = pdfFactory.createDocument();
        pdf.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excel = excelFactory.createDocument();
        excel.open();
    }
}
```

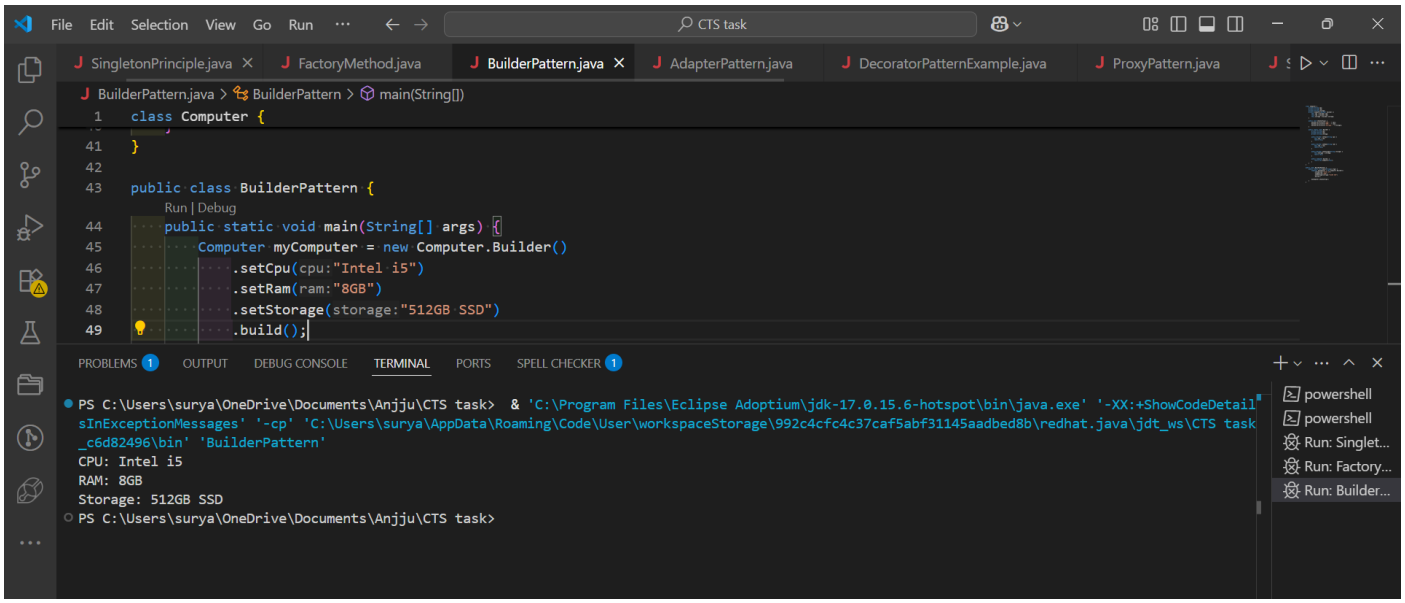
OUTPUT:



Exercise 3: Implementing the Builder Pattern

```
class Computer {
    private String cpu;
    private String ram;
    private String storage;
    private Computer(Builder builder) {
        this.cpu = builder.cpu;
        this.ram = builder.ram;
        this.storage = builder.storage;
    }

    public void showConfig() {
        System.out.println("CPU: " + cpu);
        System.out.println("RAM: " + ram);
        System.out.println("Storage: " + storage);
    }

    public static class Builder {
        private String cpu;
        private String ram;
        private String storage;

        public Builder setCpu(String cpu) {
            this.cpu = cpu;
            return this;
        }

        public Builder setRam(String ram) {
            this.ram = ram;
            return this;
        }

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

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

public class BuilderPattern {
    public static void main(String[] args) {
        Computer myComputer = new Computer.Builder()
            .setCpu("Intel i5")
            .setRam("8GB")
            .setStorage("512GB SSD")
            .build();

        myComputer.showConfig();
    }
}
```

OUTPUT:

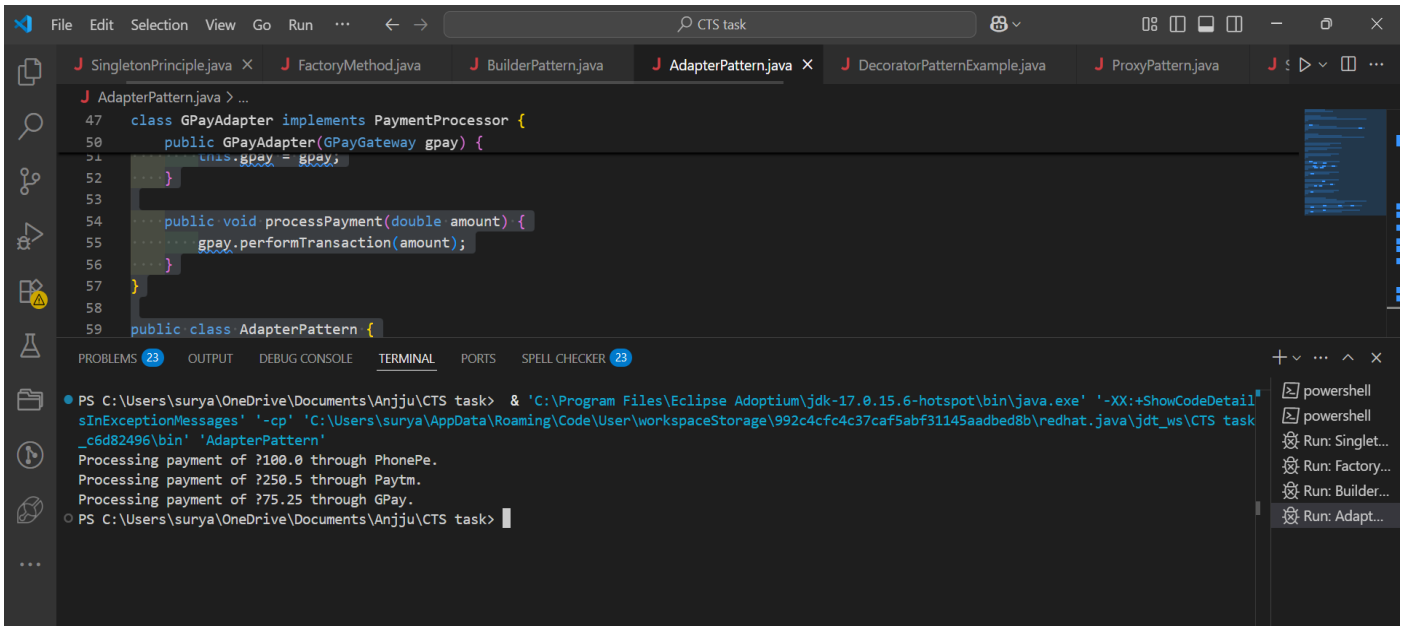


Exercise 4: Implementing the Adapter Pattern

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

class PhonePeGateway {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of ₹" + amount + " through PhonePe.");
    }
}

class PaytmGateway {
    public void makePayment(double amount) {
        System.out.println("Processing payment of ₹" + amount + " through Paytm.");
    }
}

class GPayGateway {
    public void performTransaction(double amount) {
        System.out.println("Processing payment of ₹" + amount + " through GPay.");
    }
}

class PhonePeAdapter implements PaymentProcessor {
    private PhonePeGateway phonePe;

    public PhonePeAdapter(PhonePeGateway phonePe) {
        this.phonePe = phonePe;
    }

    public void processPayment(double amount) {
        phonePe.sendPayment(amount);
    }
}

class PaytmAdapter implements PaymentProcessor {
    private PaytmGateway paytm;
```

```
    public PaytmAdapter(PaytmGateway paytm) {
        this.paytm = paytm;
    }

    public void processPayment(double amount) {
        paytm.makePayment(amount);
    }
}

class GPayAdapter implements PaymentProcessor {
    private GPayGateway gpay;

    public GPayAdapter(GPayGateway gpay) {
        this.gpay = gpay;
    }

    public void processPayment(double amount) {
        gpay.performTransaction(amount);
    }
}

public class AdapterPattern {
    public static void main(String[] args) {
        PaymentProcessor phonePeProcessor = new PhonePeAdapter(new PhonePeGateway());
        phonePeProcessor.processPayment(100.00);

        PaymentProcessor paytmProcessor = new PaytmAdapter(new PaytmGateway());
        paytmProcessor.processPayment(250.50);

        PaymentProcessor gpayProcessor = new GPayAdapter(new GPayGateway());
        gpayProcessor.processPayment(75.25);
    }
}
```

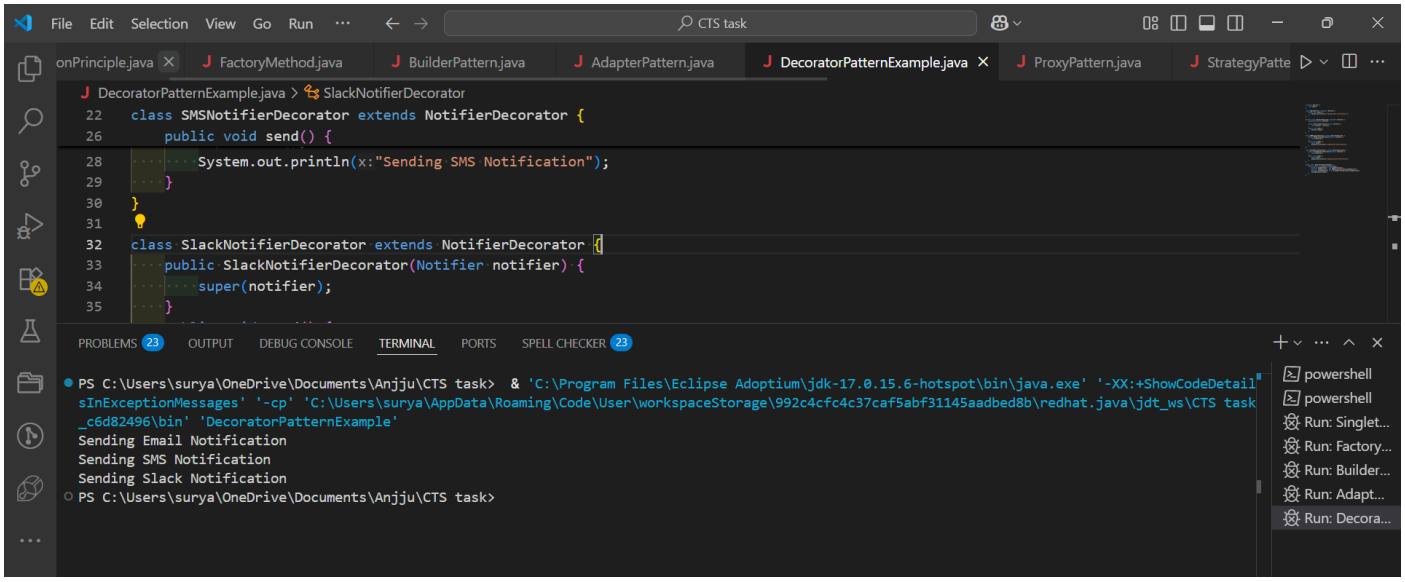
OUTPUT:



Exercise 5: Implementing the Decorator Pattern

```java
interface Notifier {
    void send();
}

class EmailNotifier implements Notifier {
    public void send() {
        System.out.println("Sending Email Notification");
    }
}

abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier;

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

class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    public void send() {
        super.send();
        System.out.println("Sending SMS Notification");
    }
}

class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    public void send() {
        super.send();
        System.out.println("Sending Slack Notification");
    }
}

public class DecoratorPatternExample {
    public static void main(String[] args) {
        Notifier baseNotifier = new EmailNotifier();
        Notifier smsDecorator = new SMSNotifierDecorator(baseNotifier);
        Notifier slackDecorator = new SlackNotifierDecorator(smsDecorator);
        slackDecorator.send();
    }
}
```

OUTPUT:

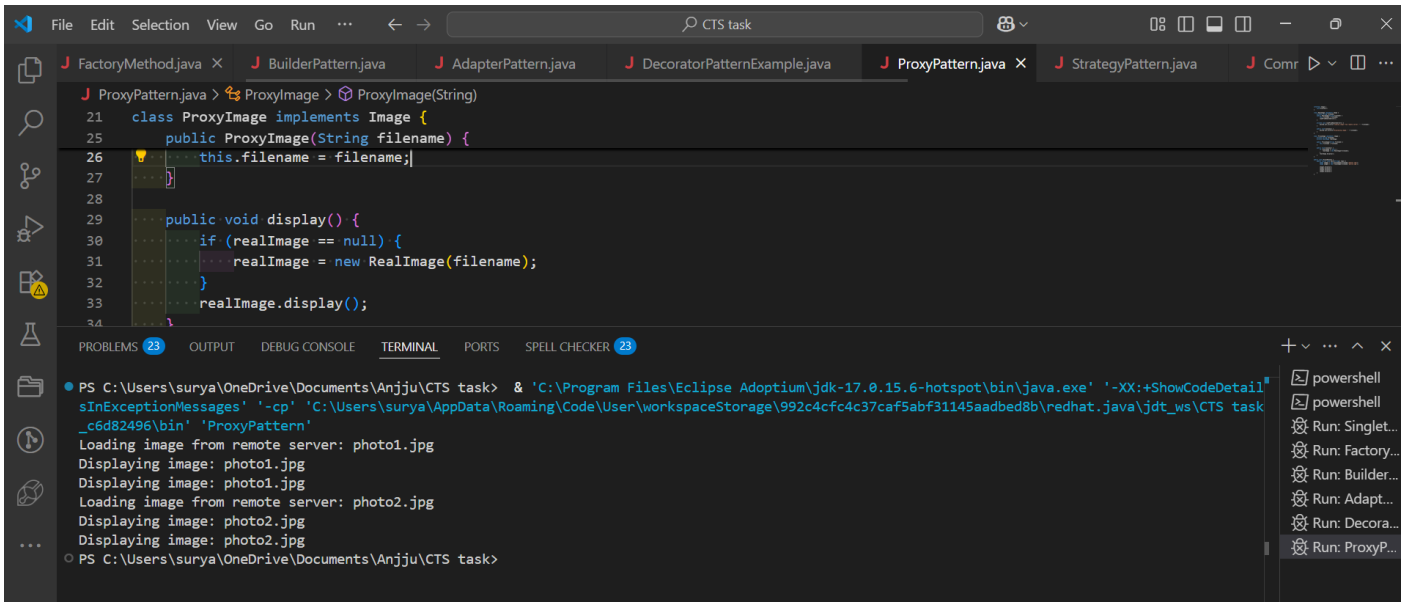


Exercise 6: Implementing the Proxy Pattern

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

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 image: " + filename);
   }
}

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);
      }
      realImage.display();
   }
}

public class ProxyPattern {
   public static void main(String[] args) {
      Image image1 = new ProxyImage("photo1.jpg");
```

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

    image1.display();
    image1.display();
    image2.display();
    image2.display();
  }
}
```

OUTPUT:

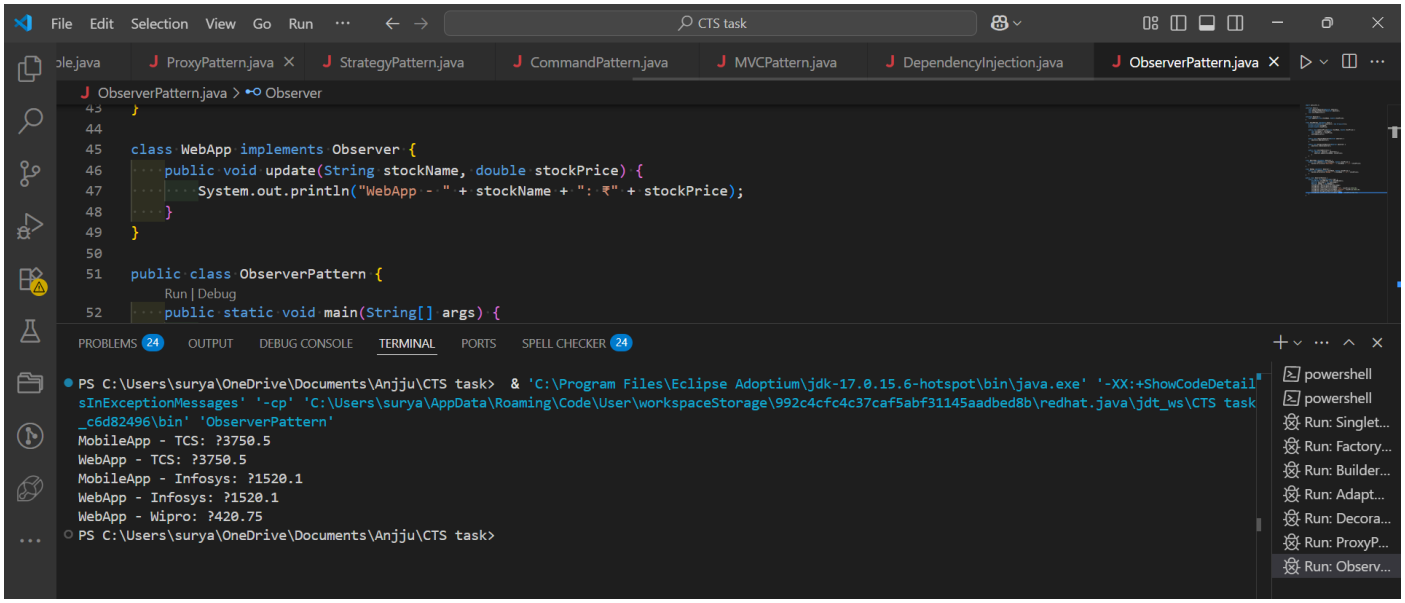


Exercise 7: Implementing the Observer Pattern

```
import java.util.*;

interface Stock {
  void registerObserver(Observer observer);
  void deregisterObserver(Observer observer);
  void notifyObservers();
}

interface Observer {
  void update(String stockName, double stockPrice);
}

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 deregisterObserver(Observer observer) {
        observers.remove(observer);
    }

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

class MobileApp implements Observer {
    public void update(String stockName, double stockPrice) {
        System.out.println("MobileApp - " + stockName + ": ₹" + stockPrice);
    }
}

class WebApp implements Observer {
    public void update(String stockName, double stockPrice) {
        System.out.println("WebApp - " + stockName + ": ₹" + stockPrice);
    }
}

public class ObserverPattern {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();
        Observer mobileApp = new MobileApp();
        Observer webApp = new WebApp();
        stockMarket.registerObserver(mobileApp);
        stockMarket.registerObserver(webApp);
        stockMarket.setStockPrice("TCS", 3750.50);
        stockMarket.setStockPrice("Infosys", 1520.10);
        stockMarket.deregisterObserver(mobileApp);
        stockMarket.setStockPrice("Wipro", 420.75);
    }
}
```

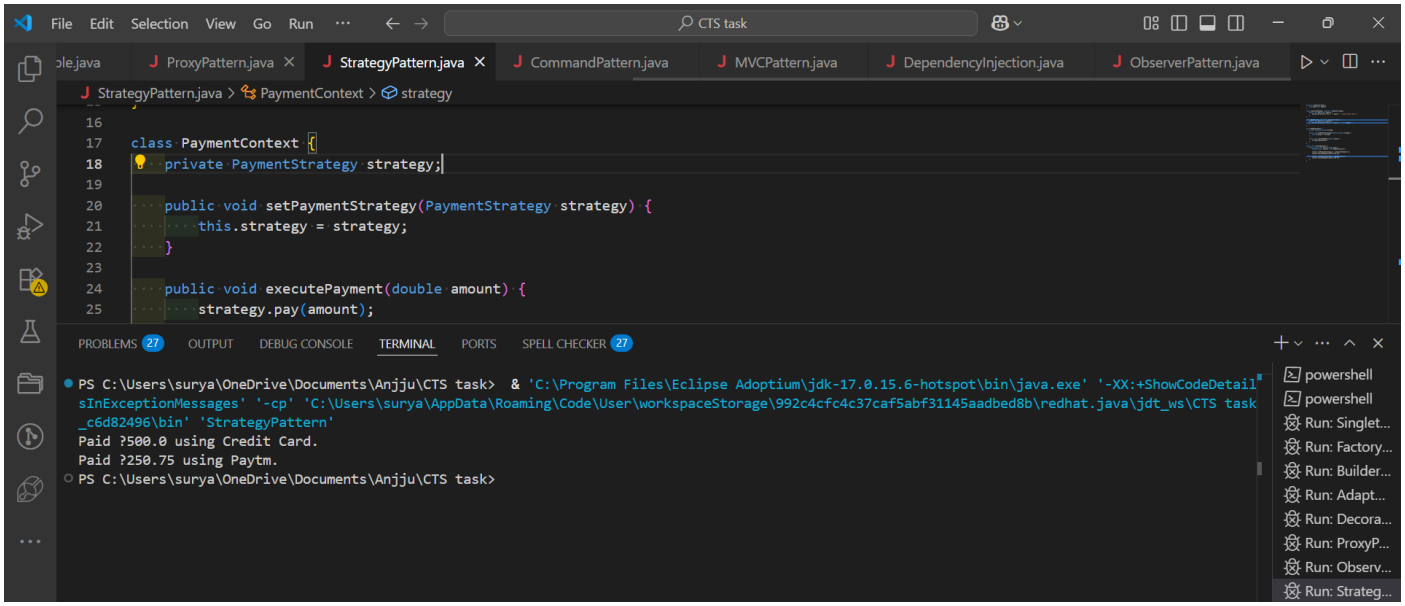
OUTPUT:



Exercise 8: Implementing the Strategy Pattern

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

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card.");
    }
}

class PaytmPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using Paytm.");
    }
}

class PaymentContext {
    private PaymentStrategy strategy;

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

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

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

        context.setPaymentStrategy(new CreditCardPayment());
        context.executePayment(500.00);

        context.setPaymentStrategy(new PaytmPayment());
        context.executePayment(250.75);
    }
}
```

OUTPUT:

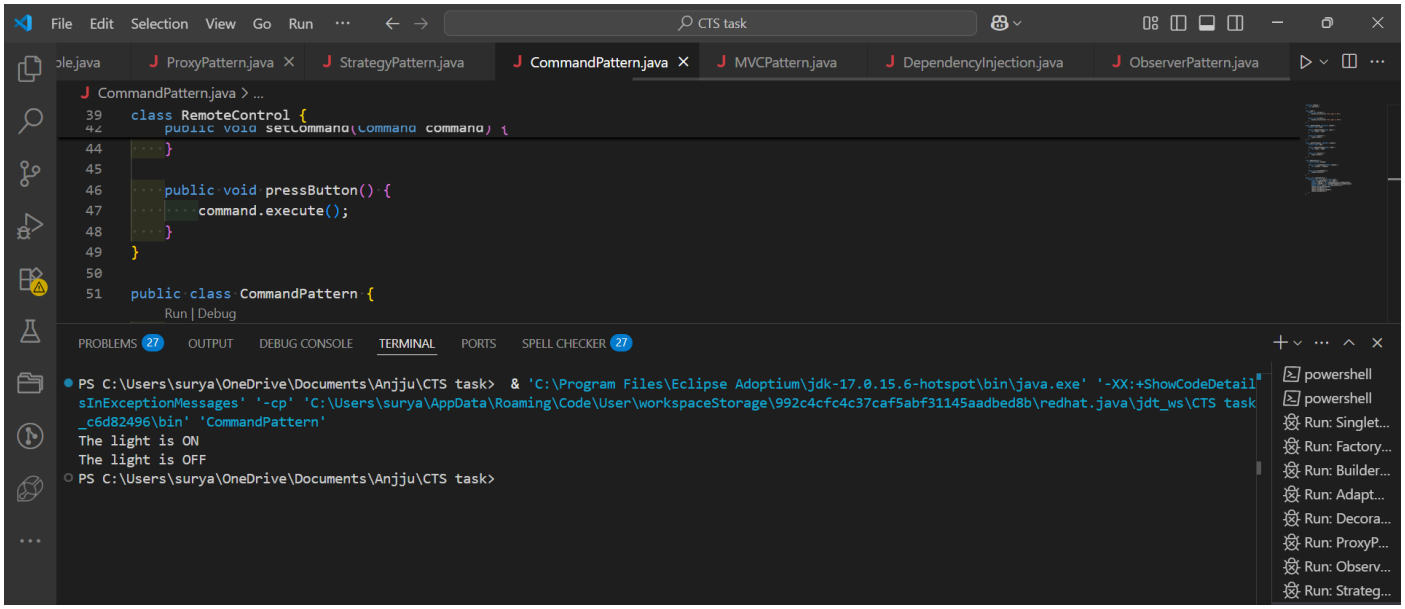


Exercise 9: Implementing the Command Pattern

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

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

    public void turnOff() {
        System.out.println("The 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 CommandPattern {
   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:

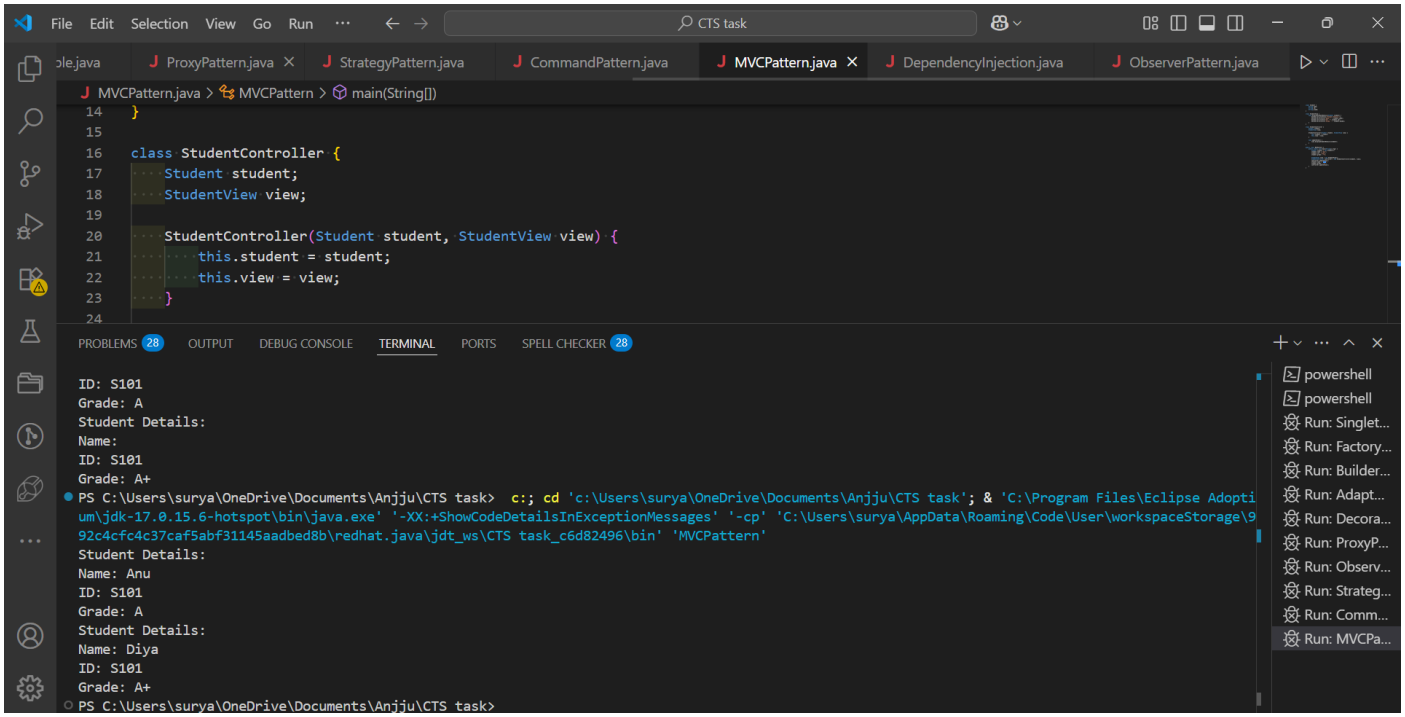


Exercise 10: Implementing the MVC Pattern

```
class Student {
   String name;
   String id;
   String grade;
}

class StudentView {
   void displayStudentDetails(Student student) {
      System.out.println("Student Details:");
```

```
        System.out.println("Name: " + student.name);
        System.out.println("ID: " + student.id);
        System.out.println("Grade: " + student.grade);
    }
}

class StudentController {
    Student student;
    StudentView view;

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

    void updateView() {
        view.displayStudentDetails(student);
    }
}

public class MVCPattern {
    public static void main(String[] args) {
        Student student = new Student();
        student.name = "Anu";
        student.id = "S101";
        student.grade = "A";

        StudentView view = new StudentView();
        StudentController controller = new StudentController(student, view);
        controller.updateView();
        student.name = "";
        student.grade = "A+";
        controller.updateView();
    }
}
```

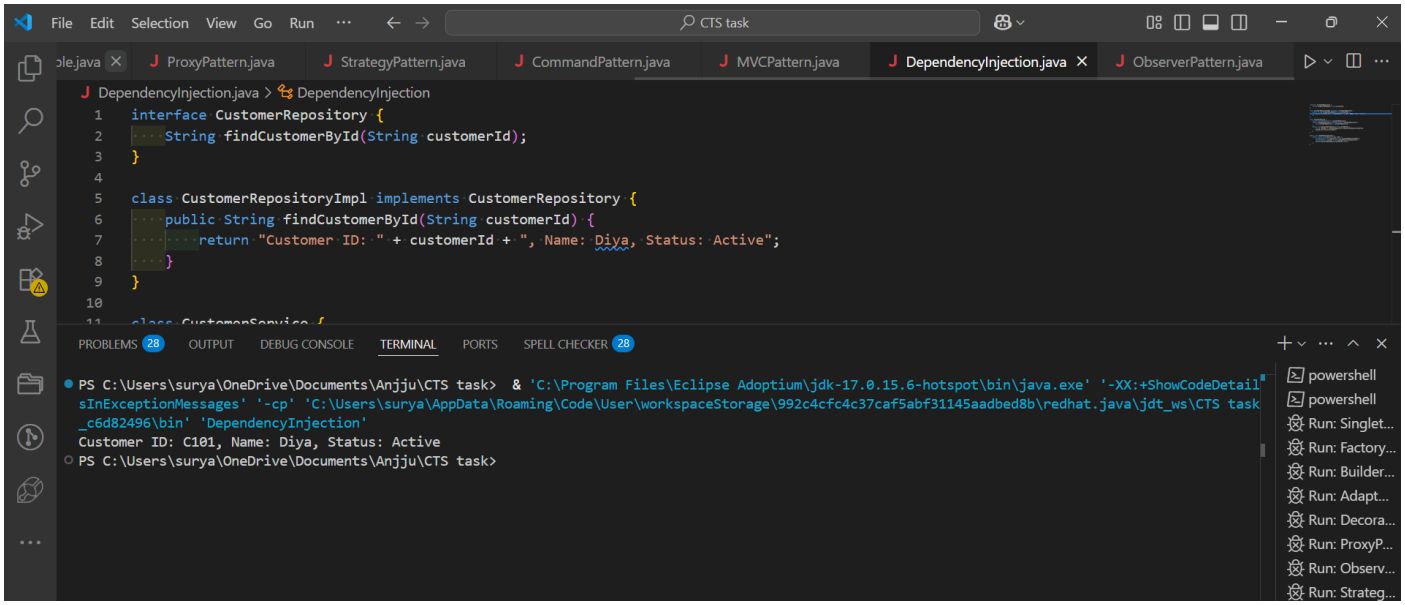
OUTPUT:



Exercise 11: Implementing Dependency Injection

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

class CustomerRepositoryImpl implements CustomerRepository {
    public String findCustomerById(String customerId) {
        return "Customer ID: " + customerId + ", Name: Diya, Status: Active";
    }
}

class CustomerService {
    private CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
    public void getCustomerDetails(String customerId) {
        String customer = customerRepository.findCustomerById(customerId);
        System.out.println(customer);
    }
}

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

OUTPUT:


File Edit Selection View Go Run
CTS task
ProxyPattern.java
StrategyPattern.java
CommandPattern.java
MVCPattern.java
DependencyInjection.java
ObserverPattern.java
DependencyInjection.java > DependencyInjection
interface CustomerRepository {
String findCustomerById(String customerId);
}
class CustomerRepositoryImpl implements CustomerRepository {
public String findCustomerById(String customerId) {
return "Customer ID: " + customerId + ", Name: Diya, Status: Active";
}
}
PROBLEMS 28
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
SPELL CHECKER 28
PS C:\Users\surya\OneDrive\Documents\Anjju\CTS task> & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.15.6-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\surya\AppData\Roaming\Code\User\workspaceStorage\992c4cfc4c37caf5abf31145aadbed8b\redhat.java\jdt_ws\CTS task_c6d82496\bin' 'DependencyInjection'
Customer ID: C101, Name: Diya, Status: Active
PS C:\Users\surya\OneDrive\Documents\Anjju\CTS task>
powershell
powershell
Run: Singlet...
Run: Factory...
Run: Builder...
Run: Adapt...
Run: Decora...
Run: ProxyP...
Run: Observ...
Run: Strateg...