**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:** You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Logger.java**

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger initialized.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

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

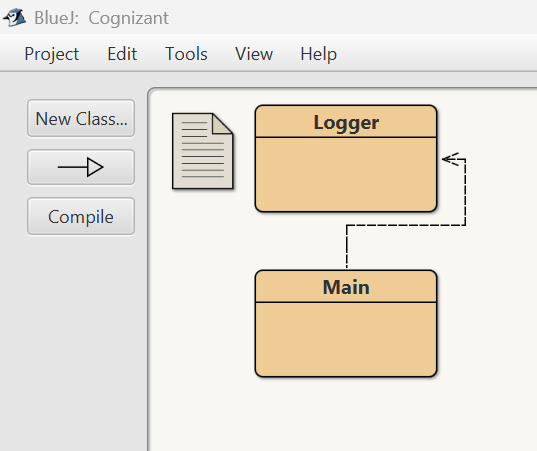
Logger logger2 = Logger.getInstance();

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

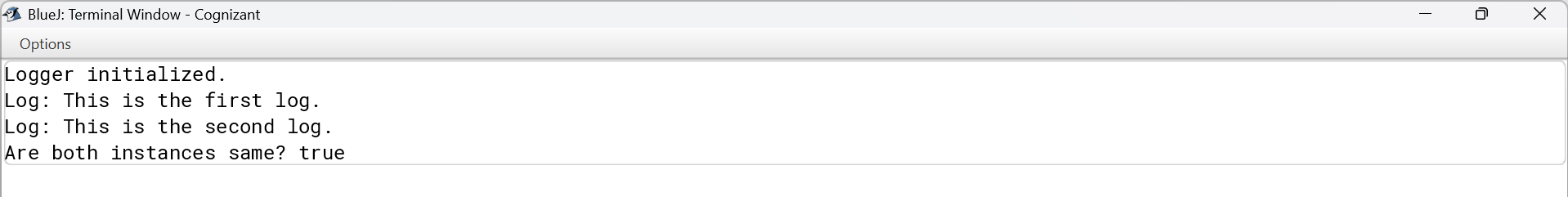
System.out.println("Are both instances same? " + (logger1 == logger2));

}

}



**Output:**



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

**Scenario:** You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**// Document.java**

interface Document {

void open();

}

**// WordDocument.java**

class WordDocument implements Document {

public void open() {

System.out.println("Opening Word Document...");

}

}

**// PdfDocument.java**

class PdfDocument implements Document {

public void open() {

System.out.println("Opening PDF Document...");

}

}

**// ExcelDocument.java**

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel Document...");

}

}

**// DocumentFactory.java**

abstract class DocumentFactory {

public abstract Document createDocument();

}

**// WordFactory.java**

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**// PdfFactory.java**

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**// ExcelFactory.java**

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**// Main.java**

public class Main {

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();

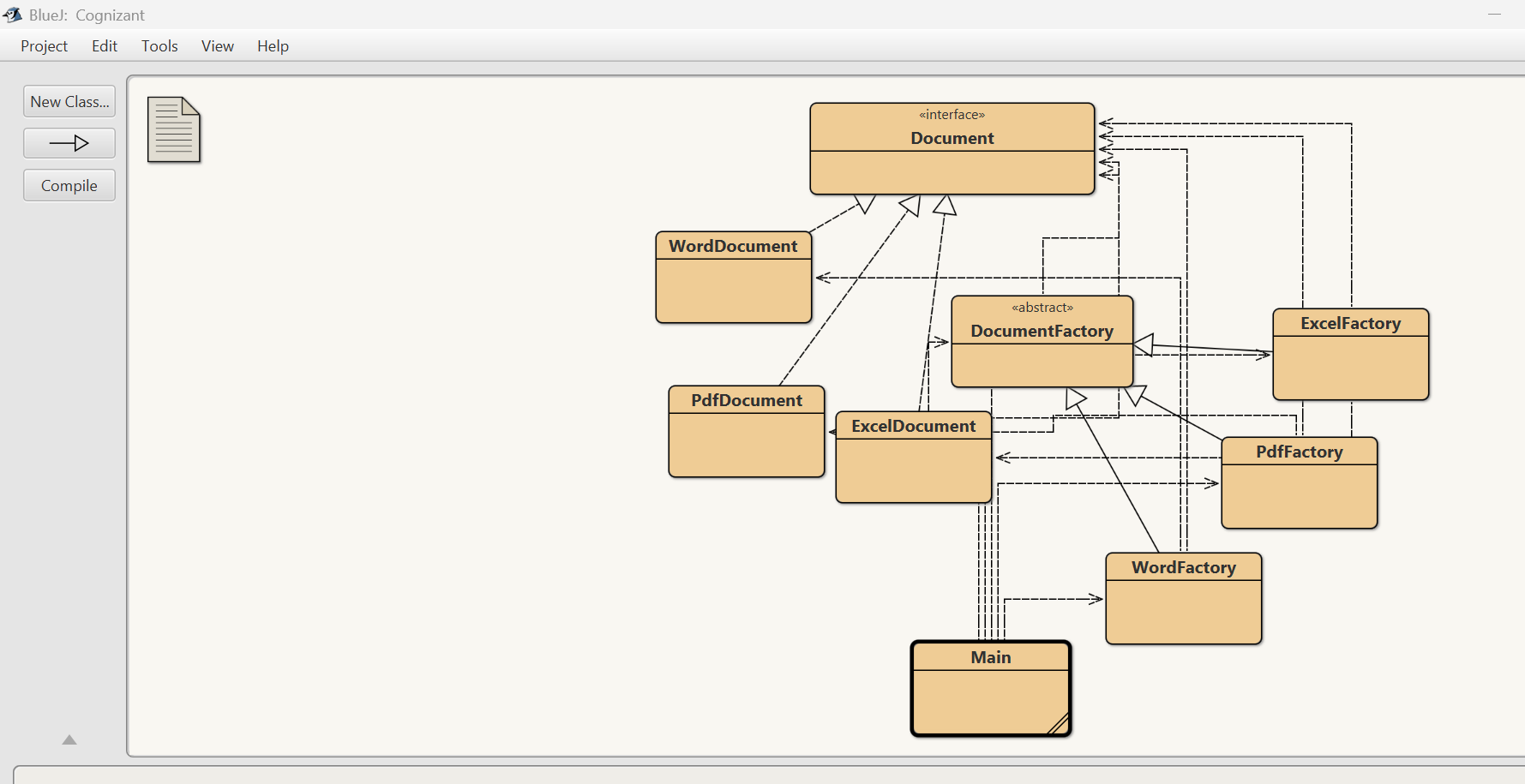
DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

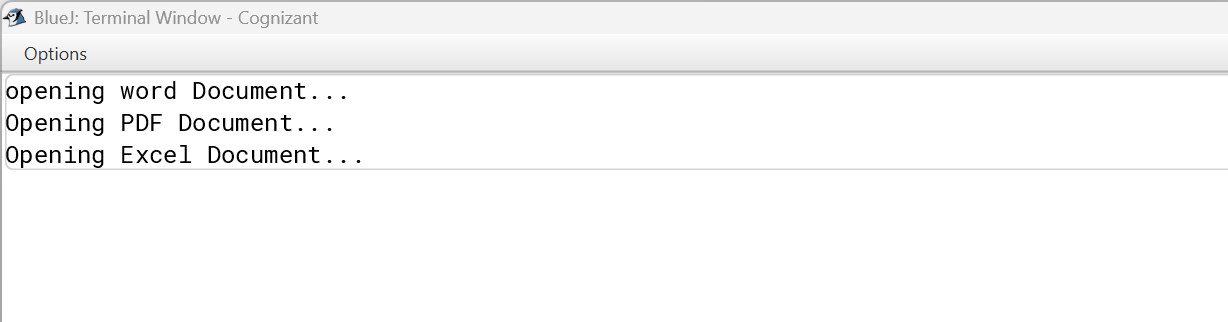
excelDoc.open();

}

}

****

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

**Scenario:** You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**// Computer.java**

public 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 displaySpecs() {

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);

}

}

}

**// Main.java**

public class Main {

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB ssd")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("500GB hdd")

.build();

System.out.println("Gaming PC Specs:");

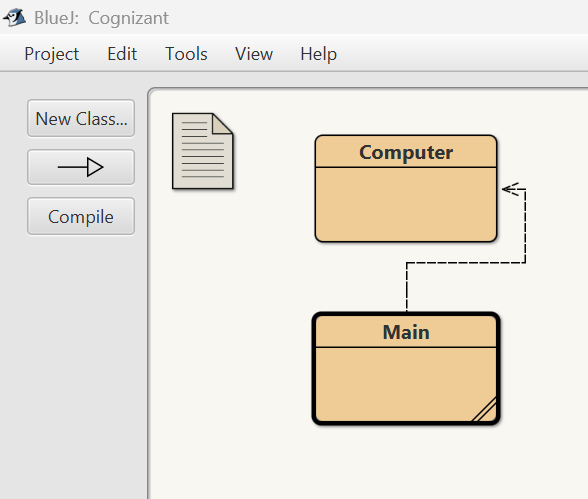
gamingPC.displaySpecs();

System.out.println("\nOffice PC Specs:");

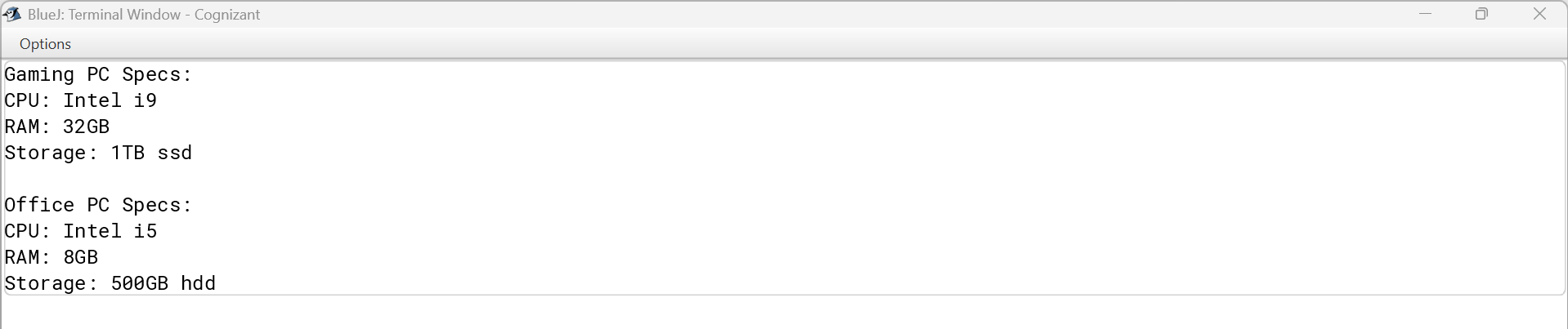
officePC.displaySpecs();

}

}



**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:** You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void sendMoney(double amount) {

System.out.println("Paying ₹" + amount + " via PayPal");

}

}

class StripeGateway {

public void makePayment(double amount) {

System.out.println("Paying ₹" + amount + " via juspay");

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal = new PayPalGateway();

public void processPayment(double amount) {

payPal.sendMoney(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe = new StripeGateway();

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

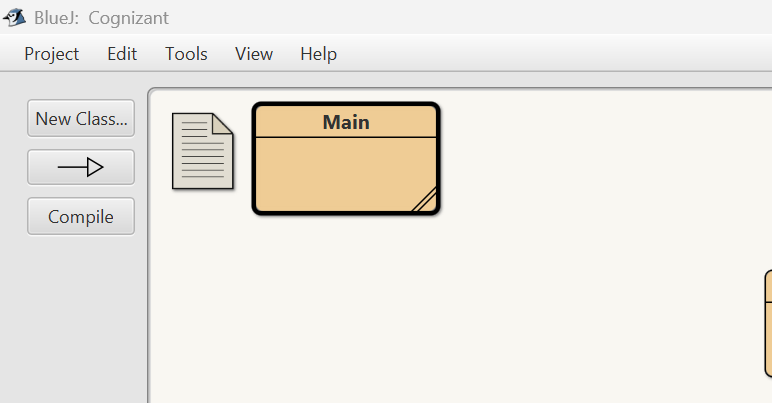
paypal.processPayment(2500);

PaymentProcessor stripe = new StripeAdapter();

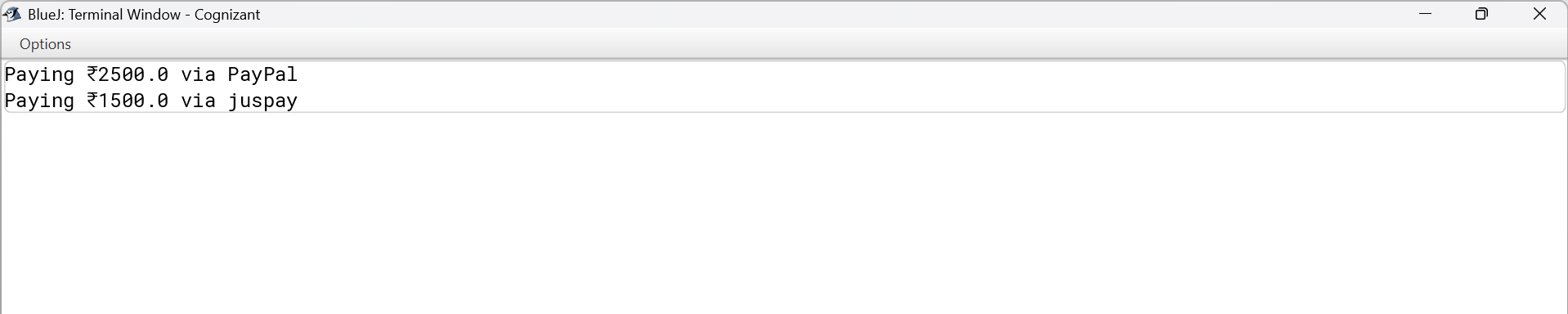
stripe.processPayment(1500);

}

}



**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:** You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

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 wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.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: " + message);

}

}

public class Main {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

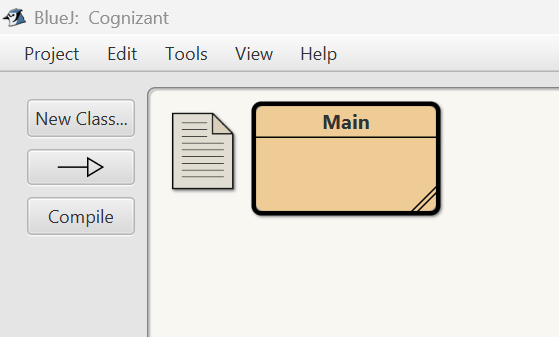
Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

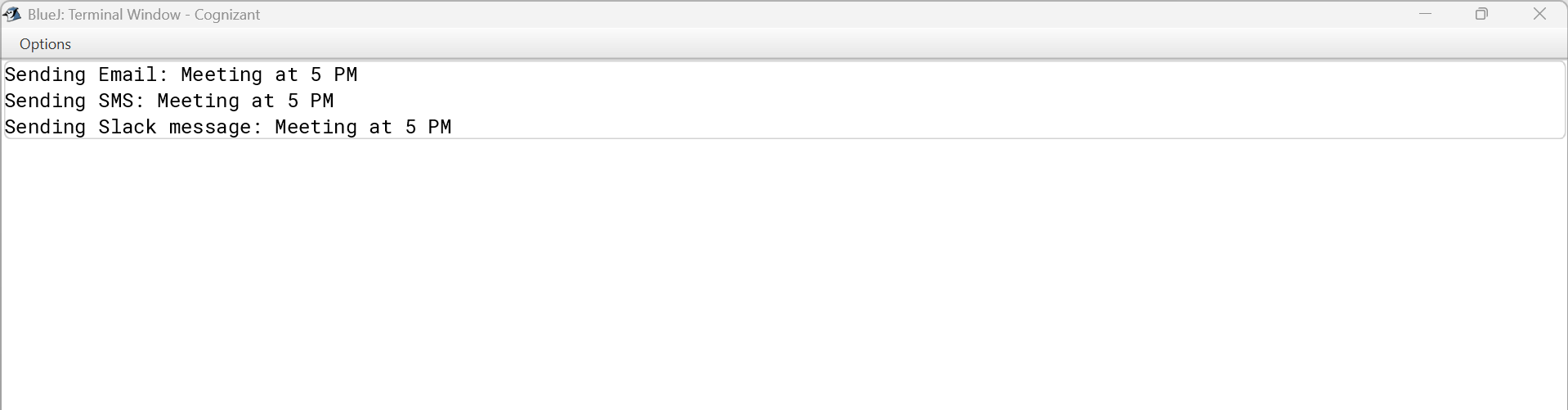
fullNotifier.send("Meeting at 5 PM");

}

}



**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario**: You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching**.**

interface Image {

void display();

}

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromServer();

}

private void loadFromServer() {

System.out.println("Loading image from server: " + fileName);

}

public void display() {

System.out.println("Displaying: " + 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 proxy {

public static void main(String[] args) {

Image img1 = new ProxyImage("photo1.jpg");

img1.display(); // Loads and displays

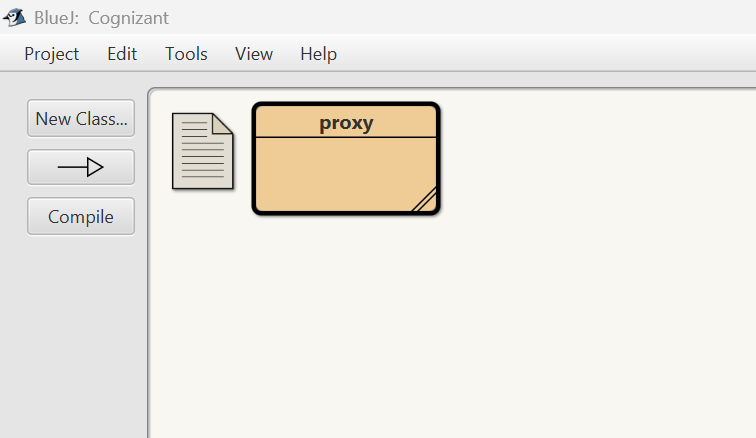
img1.display(); // Only displays

Image img2 = new ProxyImage("photo2.jpg");

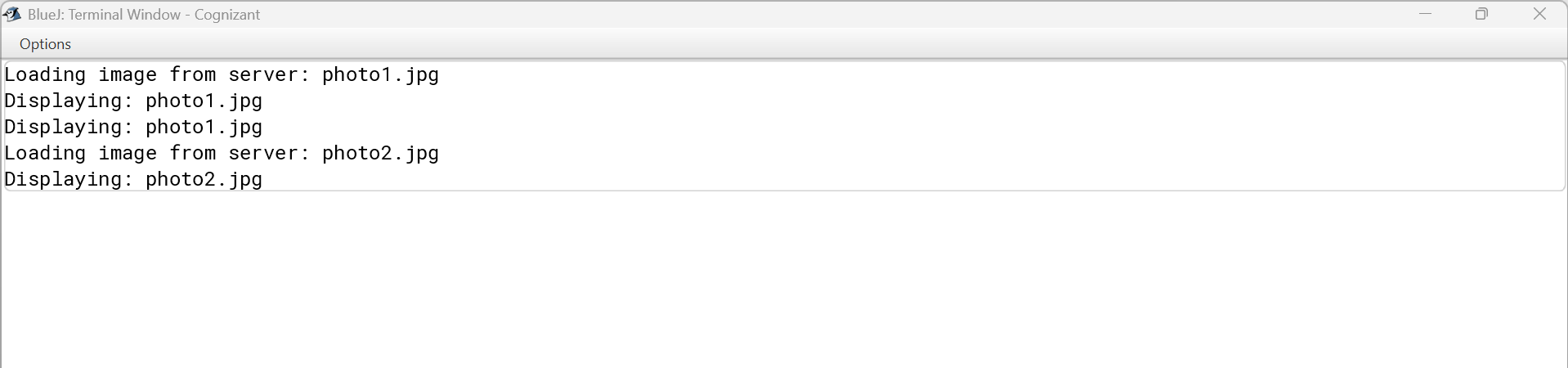
img2.display();

}

}



**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Scenario:** You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

import java.util.\*;

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private String stockName;

private double price;

public void setStockPrice(String name, double price) {

this.stockName = name;

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(stockName, price);

}

}

}

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp - " + stockName + " price updated to ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp - " + stockName + " price updated to ₹" + price);

}

}

public class observer\_pattern{

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

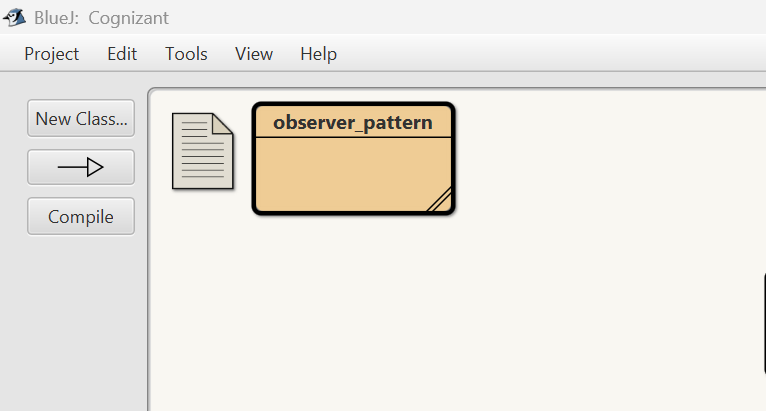
market.register(web);

market.setStockPrice("Cognizant", 3550.50);

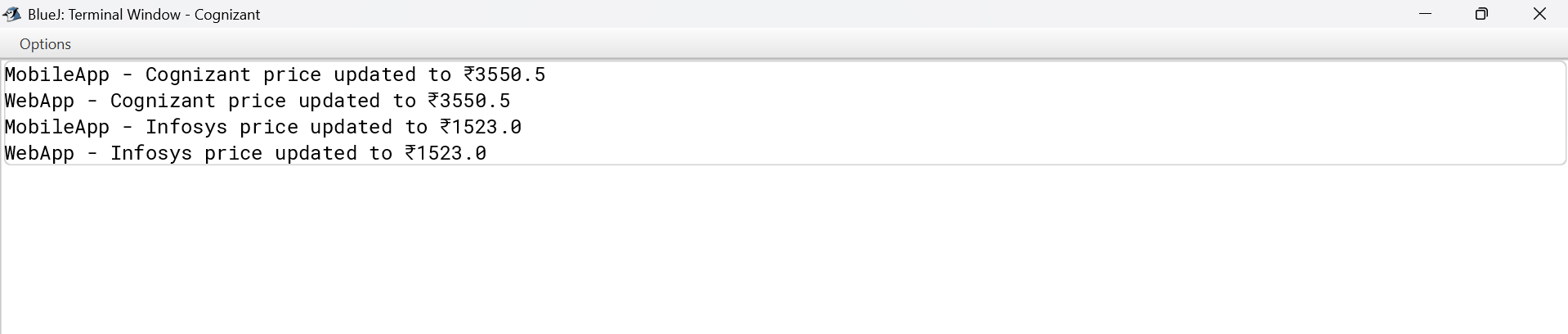
market.setStockPrice("Infosys", 1523.00);

}

}



**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:** You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

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 setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

System.out.println("No payment strategy selected.");

}

}

}

public class strategypattern {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

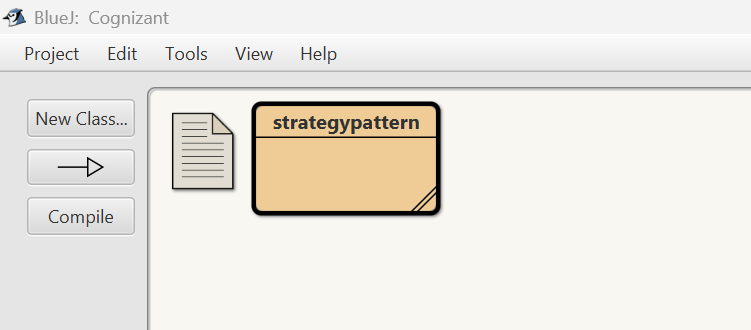
context.payAmount(3000);

context.setPaymentStrategy(new PayPalPayment());

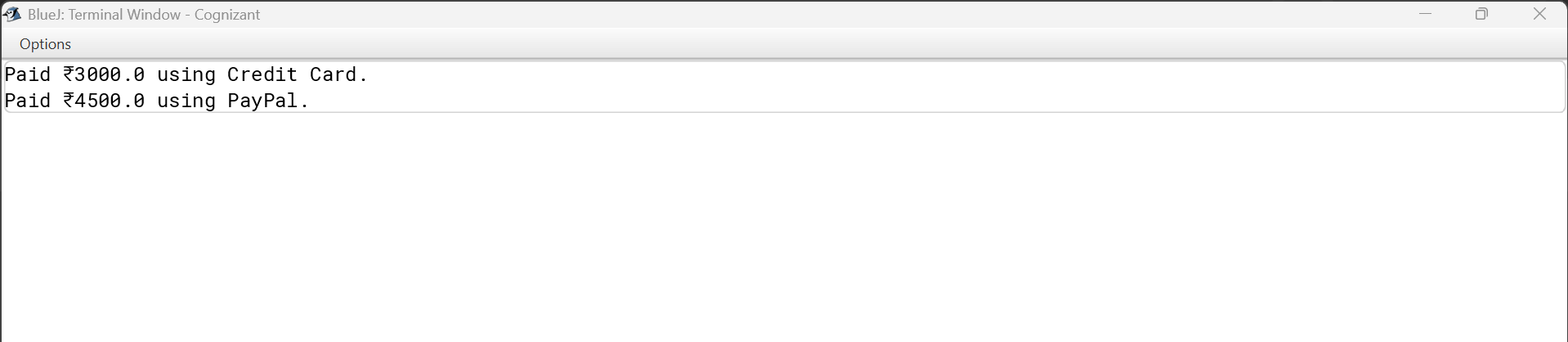
context.payAmount(4500);

}

}



**Output:**



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

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() {

if (command != null) {

command.execute();

}

}

}

public class command\_pattern {

public static void main(String[] args) {

Light light = new Light();

Command on = new LightOnCommand(light);

Command off = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(on);

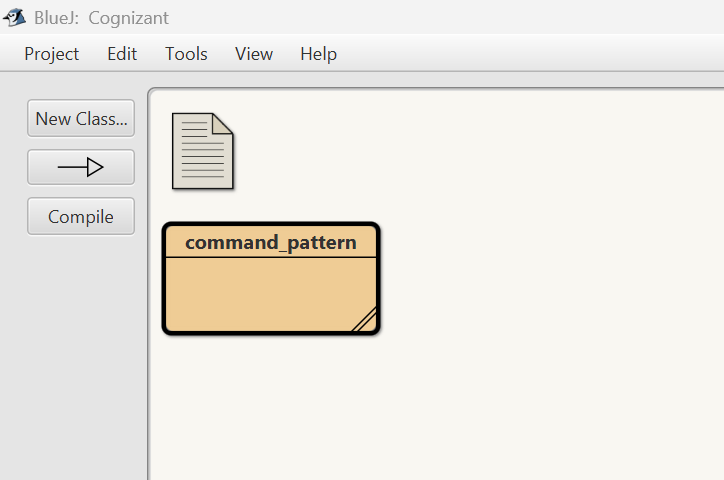
remote.pressButton();

remote.setCommand(off);

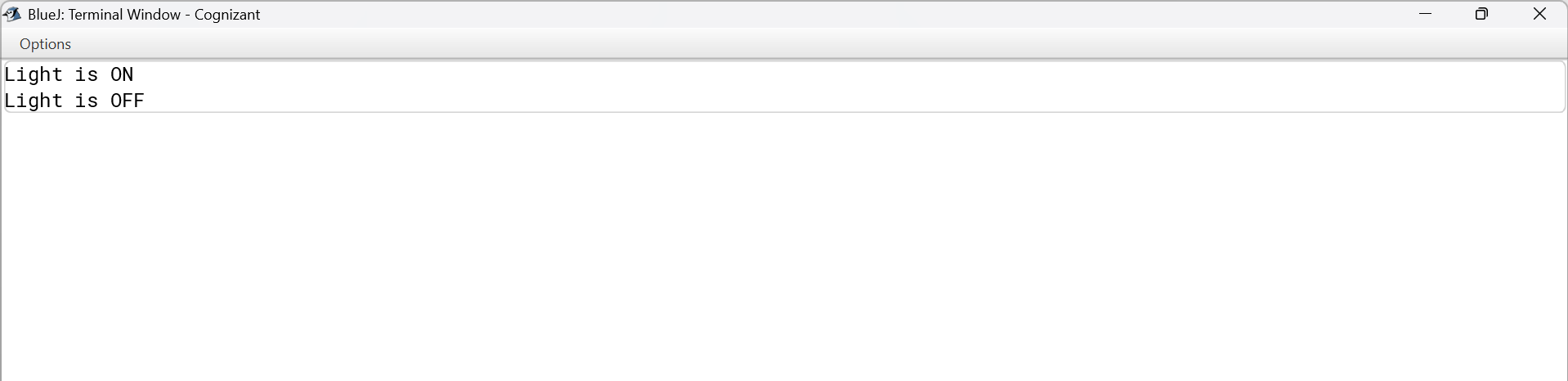
remote.pressButton();

}

}



**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:** You are developing a simple web application for managing student records using the MVC pattern.

class Student {

private String name;

private int id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public int getId() { return id; }

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 name, int id, String grade) {

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

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

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

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

}

public void setStudentName(String name) {

student.setName(name);

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

}

public class mvc {

public static void main(String[] args) {

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

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Saad");

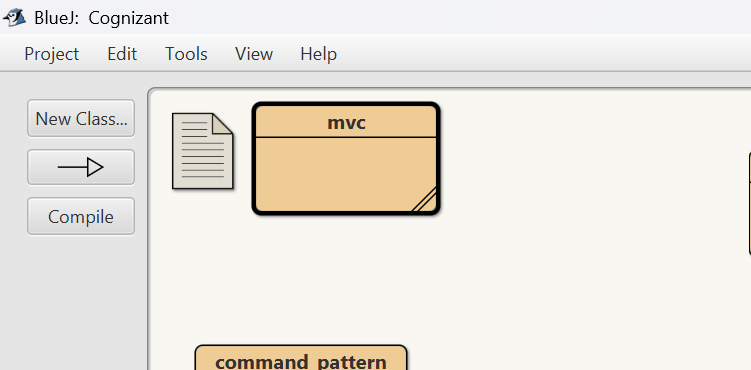
controller.setStudentGrade("A+");

System.out.println("\nAfter update:");

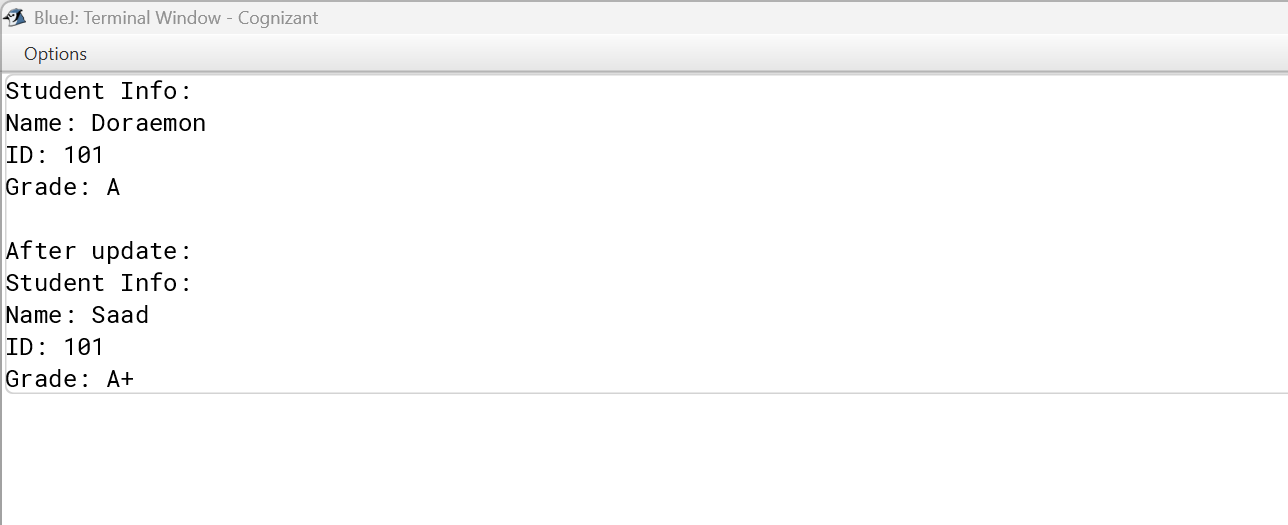
controller.updateView();

}

}

****

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:** You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer with ID: " + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void findCustomer(int id) {

String customer = repository.findCustomerById(id);

System.out.println(customer);

}

}

public class dependency\_injection{

public static void main(String[] args) {

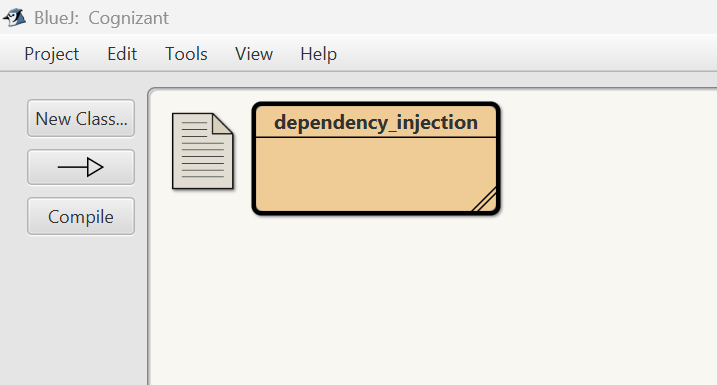
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

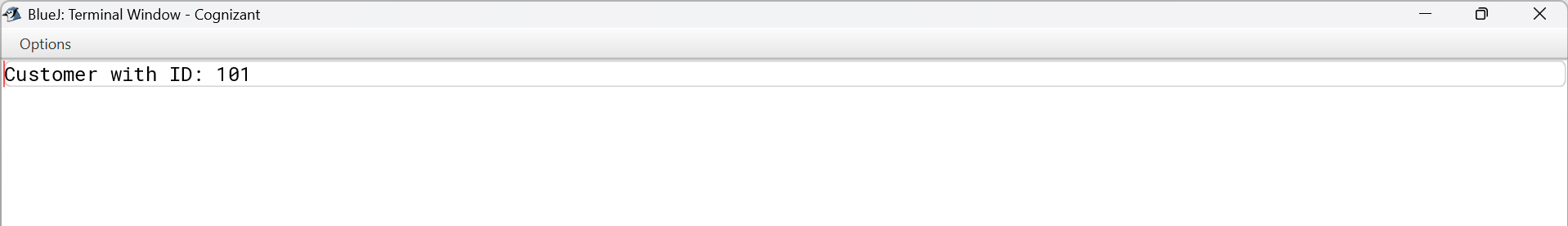
service.findCustomer(101);

}

}



**Output:**



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ThankYou\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_