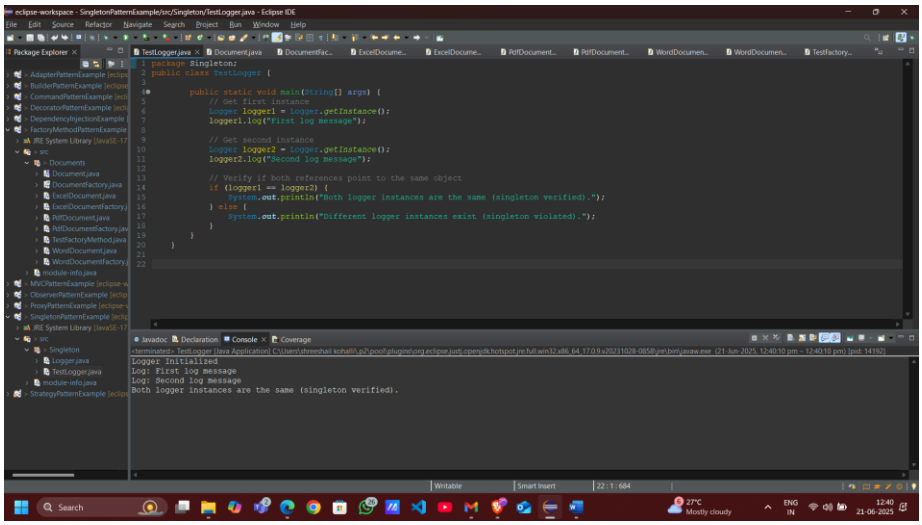
Exercise 1: Implementing the Singleton Pattern

```java
//Logger.java
package Singleton;

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);
          }
}
```

```java
//TestLogger.java
package Singleton;
public class TestLogger {

          public static void main(String[] args) {
              // Get first instance
              Logger logger1 = Logger.getInstance();
              logger1.log("First log message");

              // Get second instance
              Logger logger2 = Logger.getInstance();
              logger2.log("Second log message");

              // Verify if both references point to the same object
              if (logger1 == logger2) {
                  System.out.println("Both logger instances are the same
(singleton verified).");
              } else {
                  System.out.println("Different logger instances exist
(singleton violated).");
              }
          }
      }
```

OUTPUT:



Exercise 2: Implementing the Factory Method Pattern

```
//Document.java
package Documents;

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

```
//DocumentFactory.java
package Documents;

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

```
//ExcelDocument.java
package Documents;

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

```
//ExcelDocumentFactory.java
package Documents;

public class ExcelDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new ExcelDocument();
    }
}
```

```java
//PdfDocumentFactory.java
package Documents;

public class PdfDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new PdfDocument();
    }
}
```

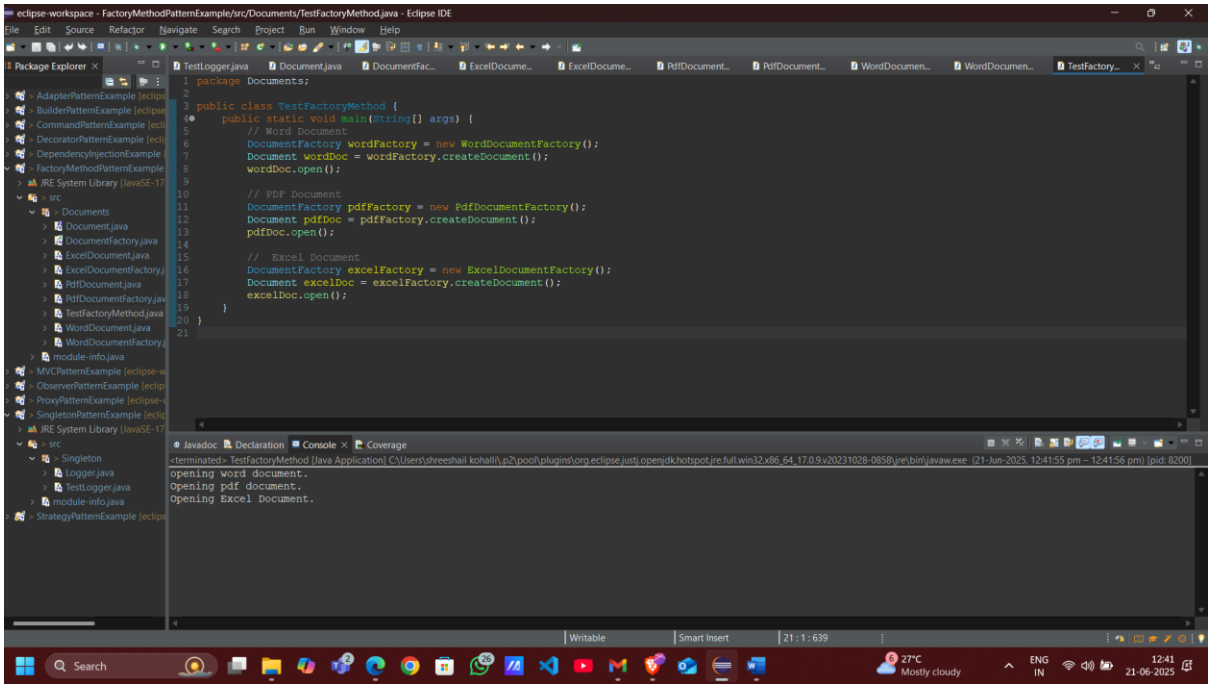
```java
//PdfDocument.java
package Documents;

public class PdfDocument implements Document {
@Override
public void open() {
      System.out.println("Opening pdf document.");
}
}
package Documents;
//wordDocument.java
public class WordDocument implements Document {
@Override
public void open() {
      System.out.println("opening word document.");
}
}
```

```java
//WordDocumentFactory.java
package Documents;

public class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}
```

```java
//TestFactoryMethod.java

package Documents;

public class TestFactoryMethod {
      public static void main(String[] args) {
        // Word Document
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();

        // PDF Document
        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();

        //  Excel Document
        DocumentFactory excelFactory = new ExcelDocumentFactory();
```

```
    Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
    }
}
```

OUTPUT:



Exercise 3: Implementing the Builder Pattern:

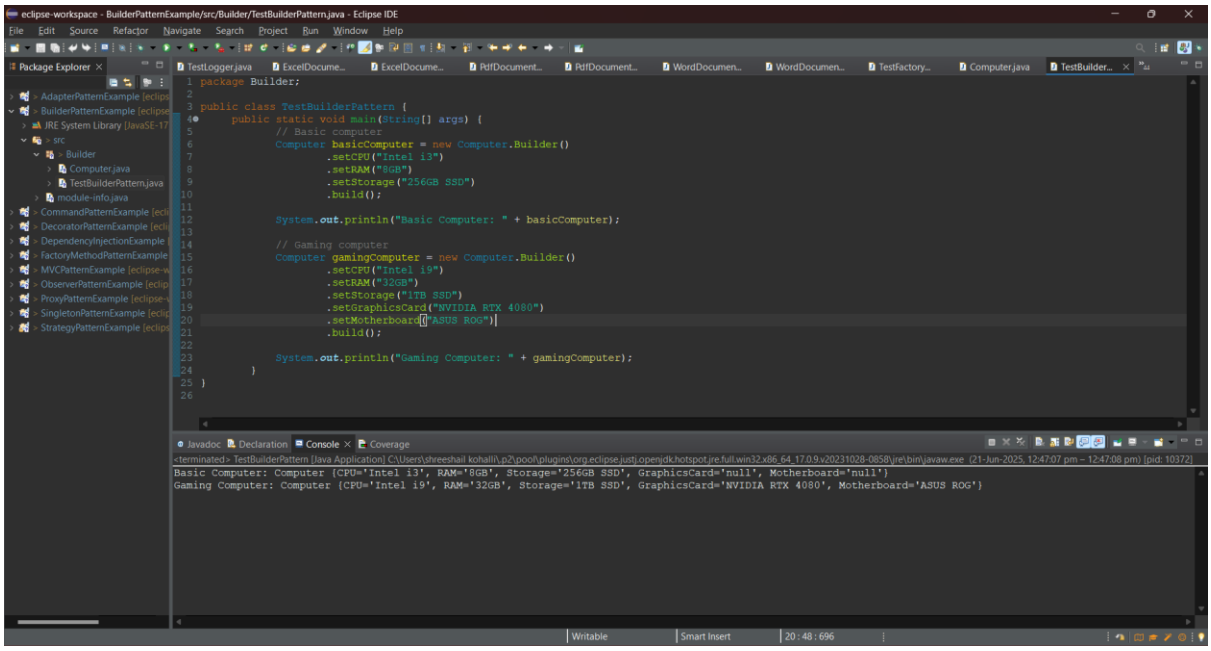
```
//Computer.java

package Builder;

public class Computer {

    private String CPU;
    private String RAM;
    private String storage;
    private String graphicsCard;
    private String motherboard;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
```

```java
        this.storage = builder.storage;
        this.graphicsCard = builder.graphicsCard;
        this.motherboard = builder.motherboard;
    }

    // Getters
    public String getCPU() { return CPU; }
    public String getRAM() { return RAM; }
    public String getStorage() { return storage; }
    public String getGraphicsCard() { return graphicsCard; }
    public String getMotherboard() { return motherboard; }

    @Override
    public String toString() {
        return "Computer {" +
                "CPU='" + CPU + '\'' +
                ", RAM='" + RAM + '\'' +
                ", Storage='" + storage + '\'' +
                ", GraphicsCard='" + graphicsCard + '\'' +
                ", Motherboard='" + motherboard + '\'' +
                '}';
    }

    // Implementing the Builder Class
    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
        private String graphicsCard;
        private String motherboard;

        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 setMotherboard(String motherboard) {
            this.motherboard = motherboard;
            return this;
        }

        // Returning the built object
        public Computer build() {
            return new Computer(this);
        }
    }
```

```
}
```

```java
//TestBuilderPattern.java
package Builder;

public class TestBuilderPattern {
	public static void main(String[] args) {
		// Basic computer
		Computer basicComputer = new Computer.Builder()
			.setCPU("Intel i3")
			.setRAM("8GB")
			.setStorage("256GB SSD")
			.build();

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

		// Gaming computer
		Computer gamingComputer = new Computer.Builder()
			.setCPU("Intel i9")
			.setRAM("32GB")
			.setStorage("1TB SSD")
			.setGraphicsCard("NVIDIA RTX 4080")
			.setMotherboard("ASUS ROG")
			.build();

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

OUTPUT:



Exercise 4: Implementing the Adapter Pattern:

```
//PaymentProcessor.java

package adapter;

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

```
//PaypalGateway.java
package adapter;

public class PaypalGateway {
      public void makePayment(double amount) {
        System.out.println("Payment of $" + amount + " processed using
PayPal.");
    }
}
```

```
//PaypalAdapter.java
package adapter;

public class PaypalAdapter implements PaymentProcessor {
    private PaypalGateway payPalGateway;

    public PaypalAdapter(PaypalGateway gateway) {
        this.payPalGateway = gateway;
    }

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

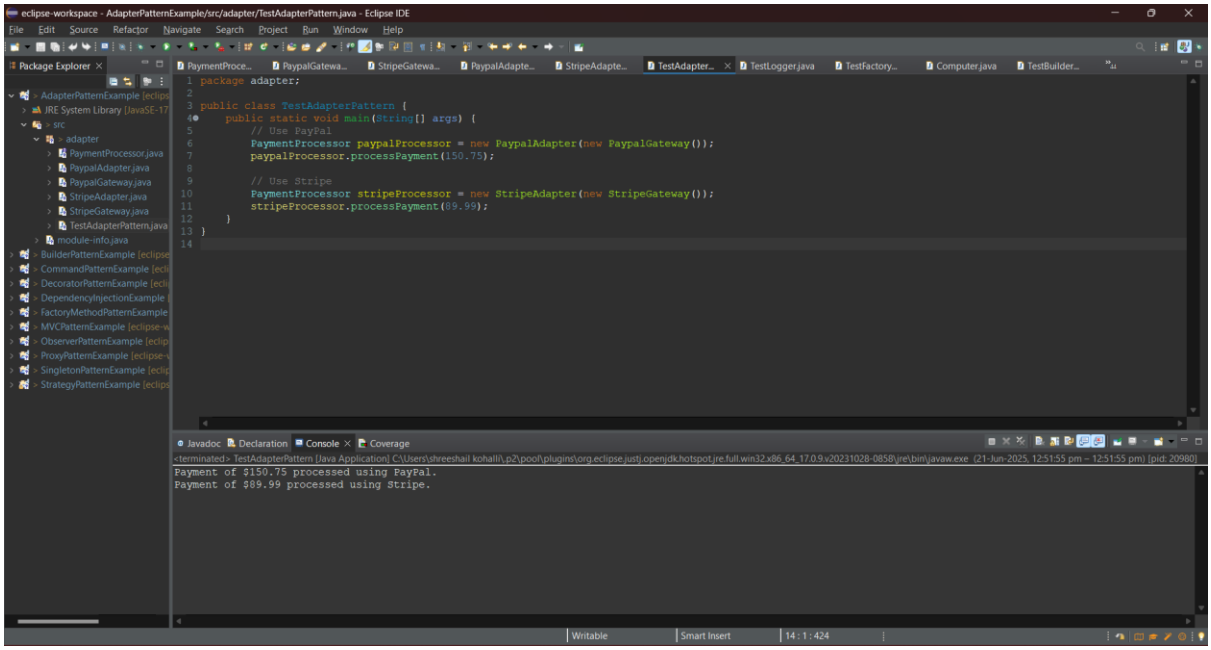
```
//StripeGateway.java
package adapter;

public class StripeGateway {
      public void pay(double amount) {
        System.out.println("Payment of $" + amount + " processed using
Stripe.");
    }
}
```

```
package adapter;

public class StripeAdapter implements PaymentProcessor {
    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway gateway) {
        this.stripeGateway = gateway;
    }

    @Override
```

```java
    public void processPayment(double amount) {
        stripeGateway.pay(amount);
    }
}
```

```java
//TestAdapterPattern.java
package adapter;

public class TestAdapterPattern {
    public static void main(String[] args) {
        // Use PayPal
        PaymentProcessor paypalProcessor = new PaypalAdapter(new
PaypalGateway());
        paypalProcessor.processPayment(150.75);

        // Use Stripe
        PaymentProcessor stripeProcessor = new StripeAdapter(new
StripeGateway());
        stripeProcessor.processPayment(89.99);
    }
}
```

OUTPUT:



Exercise 5: Implementing the Decorator Pattern

```java
//Notifier.java

package decorator;

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

```java
//NotifierDecorator.java
package decorator;

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);  // Delegate base send
    }
}
```

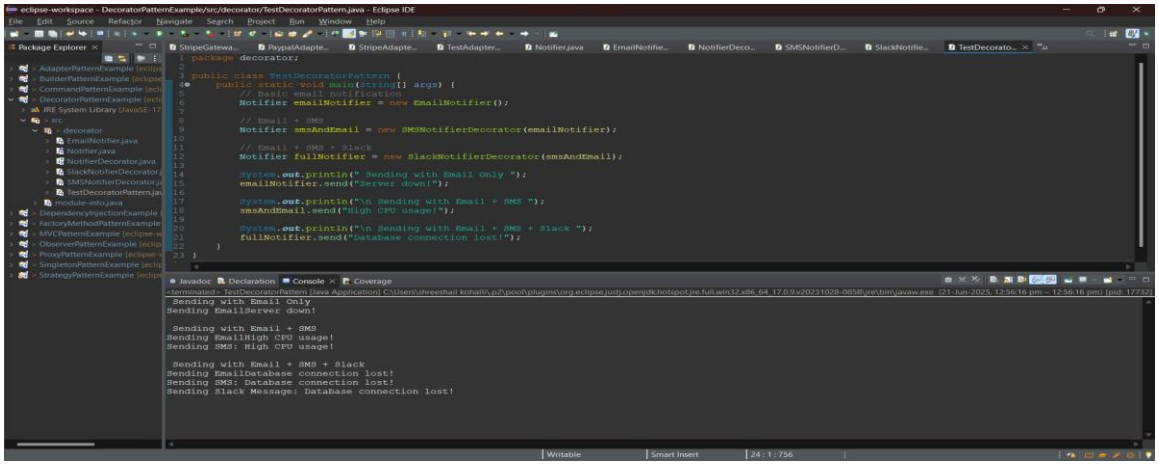
```java
//EmailNotifier.java
package decorator;

public class EmailNotifier implements Notifier {

      @Override
      public void send(String message) {
            System.out.println("Sending Email" + message);
      }
}
```

```java
//SMSNotifierDecorator.java

package decorator;

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    @Override
    public void send(String message) {
        super.send(message);  // Send original
        sendSMS(message);     // Add SMS
    }

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

```java
// SlackNotifierDecorator.java
package decorator;

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    @Override
    public void send(String message) {
        super.send(message);    // Send previous
        sendSlack(message);     // Add Slack
    }

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

```java
package decorator;

public class TestDecoratorPattern {
    public static void main(String[] args) {
        // Basic email notification
        Notifier emailNotifier = new EmailNotifier();

        // Email + SMS
        Notifier smsAndEmail = new SMSNotifierDecorator(emailNotifier);

        // Email + SMS + Slack
        Notifier fullNotifier = new SlackNotifierDecorator(smsAndEmail);

        System.out.println(" Sending with Email Only ");
        emailNotifier.send("Server down!");

        System.out.println("\n Sending with Email + SMS ");
        smsAndEmail.send("High CPU usage!");

        System.out.println("\n Sending with Email + SMS + Slack ");
        fullNotifier.send("Database connection lost!");
    }
}
```

OUTPUT:



Exercise 6: Implementing the Proxy Pattern

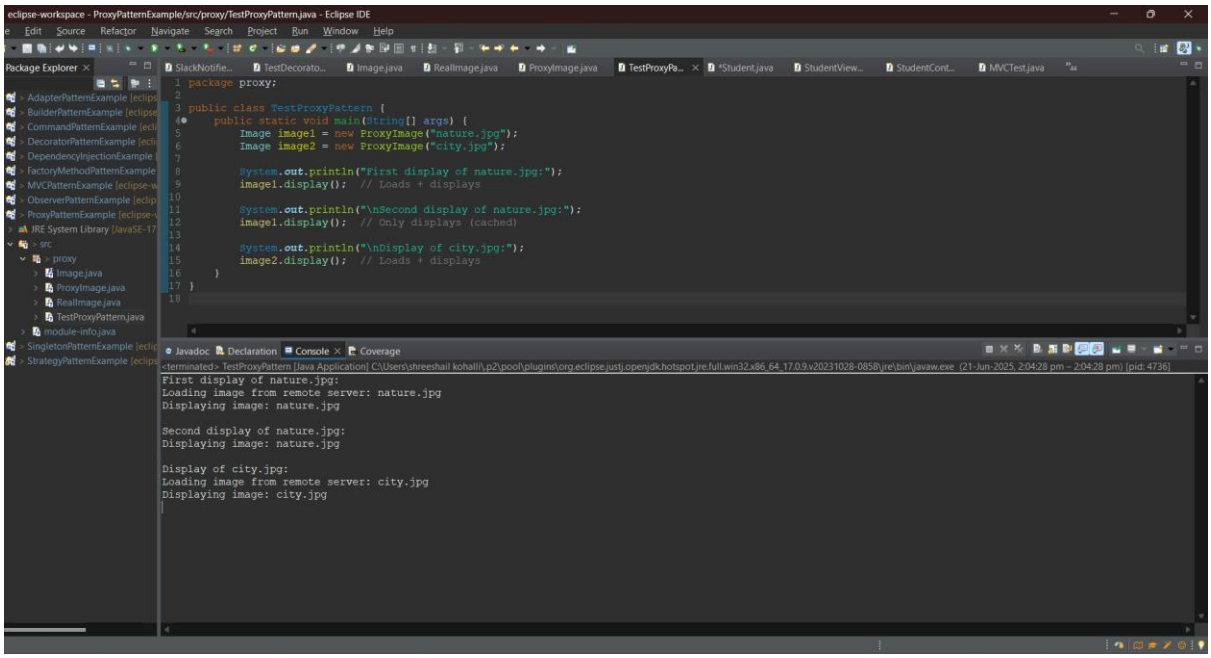
```java
//Image.java
package proxy;

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

```java
//ProxyImage.java
package proxy;

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

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

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

```java
//RealImage.java
package proxy;

public class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromRemoteServer(); // Simulate loading
    }

    private void loadFromRemoteServer() {
        System.out.println("Loading image from remote server: " +
filename);
    }

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

```java
//TestProxyPattern.java
package proxy;

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

        System.out.println("First display of nature.jpg:");
        image1.display();  // Loads + displays

        System.out.println("\nSecond display of nature.jpg:");
        image1.display();  // Only displays (cached)

        System.out.println("\nDisplay of city.jpg:");
        image2.display();  // Loads + displays
    }
}
```

OUTPUT:



Exercise 7: Implementing the Observer Pattern

```java
//Observer.java
package observer;

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

```java
//Stock.java
package observer;
public interface Stock {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();}
```

```java
//StockMarket.java
package observer;

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 name, double price) {
        this.stockName = name;
        this.stockPrice = price;
        notifyObservers(); // Notify when stock price changes
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(stockName, stockPrice);
        }
    }
}
```

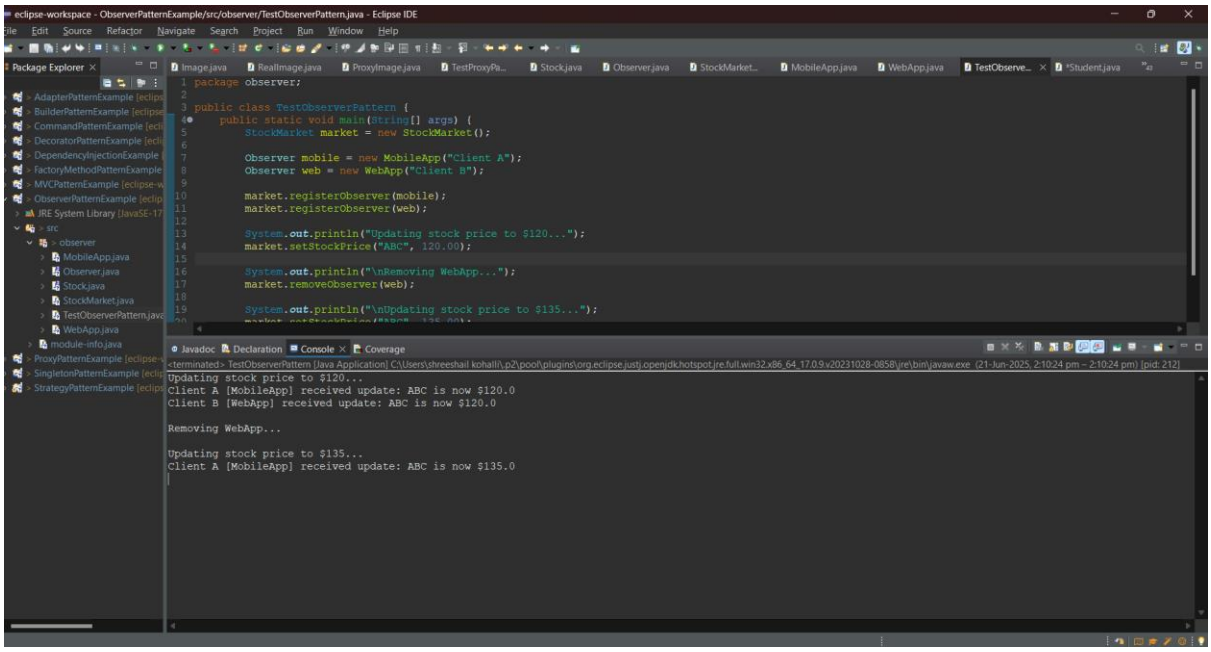
```java
MobileApp.java
package observer;

public class MobileApp implements Observer {
    private String name;

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

    @Override
    public void update(String stockName, double stockPrice) {
        System.out.println(name + " [MobileApp] received update: " +
                stockName + " is now $" + stockPrice);
    }
}
```

```java
package observer;

public class WebApp implements Observer {
    private String name;

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

```java
    }

    @Override
    public void update(String stockName, double stockPrice) {
        System.out.println(name + " [WebApp] received update: " +
                stockName + " is now $" + stockPrice);
    }
}
```

```java
//TestObserverPattern.java
package observer;

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

        Observer mobile = new MobileApp("Client A");
        Observer web = new WebApp("Client B");

        market.registerObserver(mobile);
        market.registerObserver(web);

        System.out.println("Updating stock price to $120...");
        market.setStockPrice("ABC", 120.00);

        System.out.println("\nRemoving WebApp...");
        market.removeObserver(web);

        System.out.println("\nUpdating stock price to $135...");
        market.setStockPrice("ABC", 135.00);
    }
}
```

OUTPUT:



Exercise 8: Implementing the Strategy Pattern

```java
package strategy;

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

```java
package strategy;

public class Paymentcontext {
    private PaymentStrategy strategy;

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

    public void payAmount(double amount) {
        if (strategy == null) {
            System.out.println("No payment strategy selected.");
        } else {
            strategy.pay(amount);
        }
    }
}
```

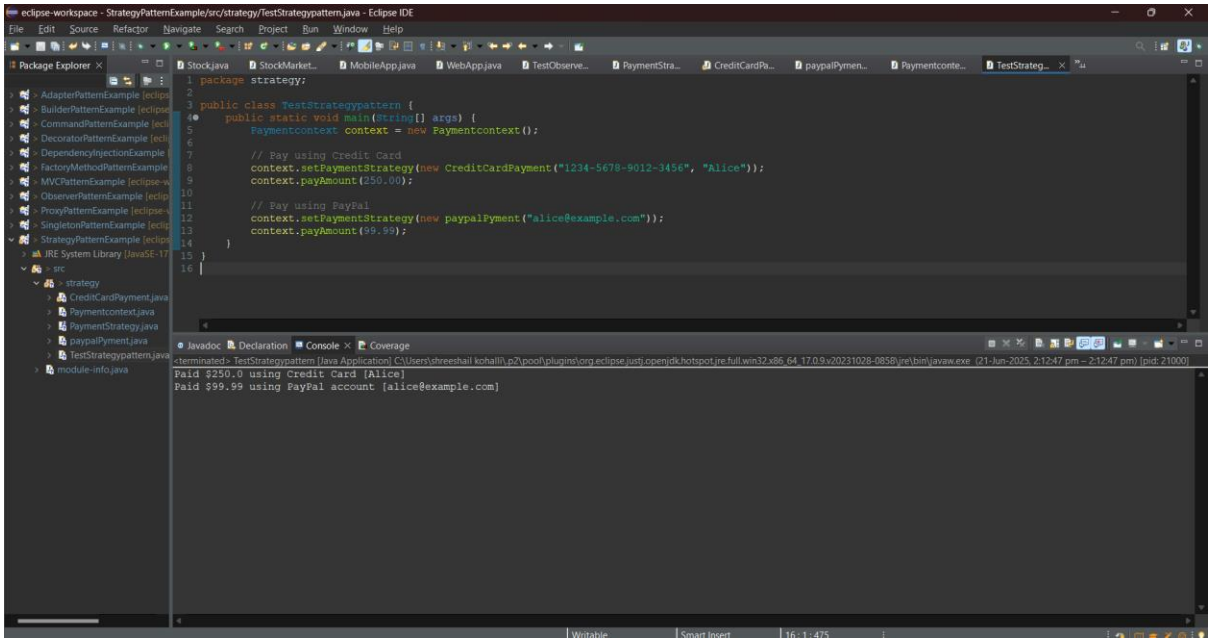
```java
package strategy;

public class paypalPyment implements PaymentStrategy {
    private String email;

    public paypalPyment(String email) {
        this.email = email;
    }

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

```java
package strategy;

public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cardHolder;

    public CreditCardPayment(String cardNumber, String cardHolder) {
        this.cardNumber = cardNumber;
        this.cardHolder = cardHolder;
    }

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

```
    }
}
```

```
package strategy;

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

        // Pay using Credit Card
        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-
3456", "Alice"));
        context.payAmount(250.00);

        // Pay using PayPal
        context.setPaymentStrategy(new paypalPyment("alice@example.com"));
        context.payAmount(99.99);
    }
}
```

OUTPUT:



Exercise 9: Implementing the Command Pattern

```java
package command;

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

```java
package command;

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

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}
```

```java
package command;

public class LightOnCommand implements Command {
    private Light light;

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

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

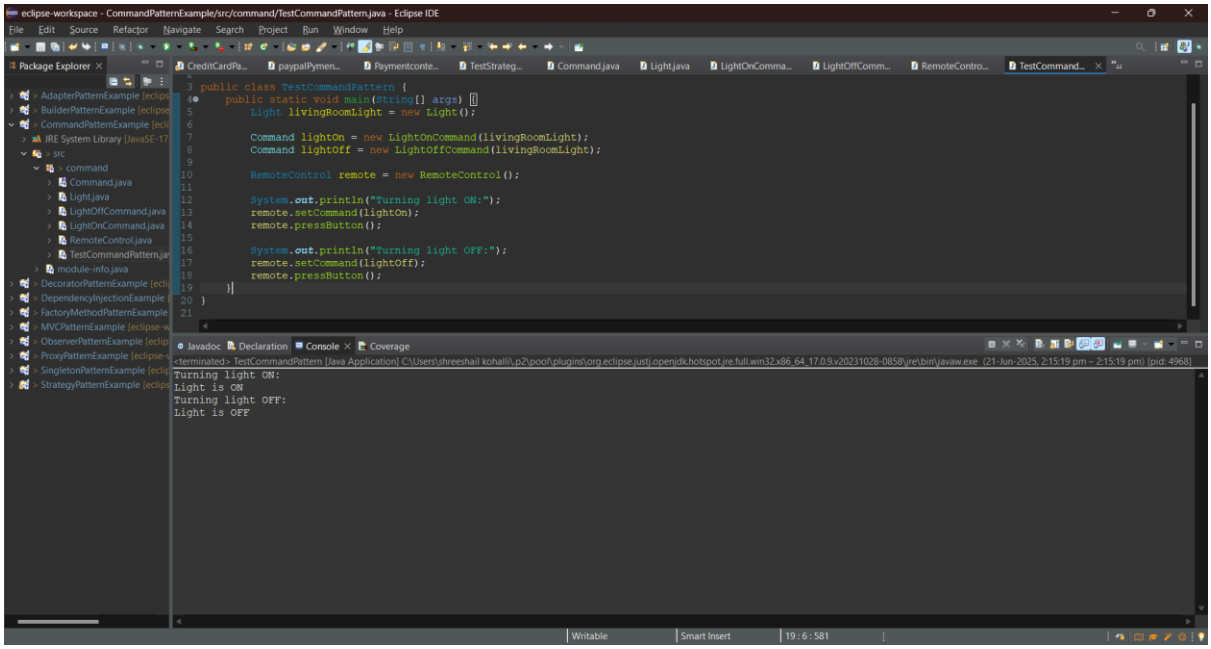
```java
package command;

public class LightOffCommand implements Command {
    private Light light;

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

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

```java
package command;

public class RemoteControl {
    private Command command;
```

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

    public void pressButton() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println("No command set");
        }
    }
}
```

```java
package command;

public class TestCommandPattern {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        System.out.println("Turning light ON:");
        remote.setCommand(lightOn);
        remote.pressButton();

        System.out.println("Turning light OFF:");
        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

OUTPUT:



Exercise 10: Implementing the MVC Pattern

```java
package mvc;

public class Student {
    private String id;
    private String name;
    private String grade;

    // Constructor
    public Student(String id, String name, String grade) {
        this.id = id;
        this.name = name;
        this.grade = grade;
    }

    // Getters & Setters
    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

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

    public String getGrade() {
        return grade;
    }

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

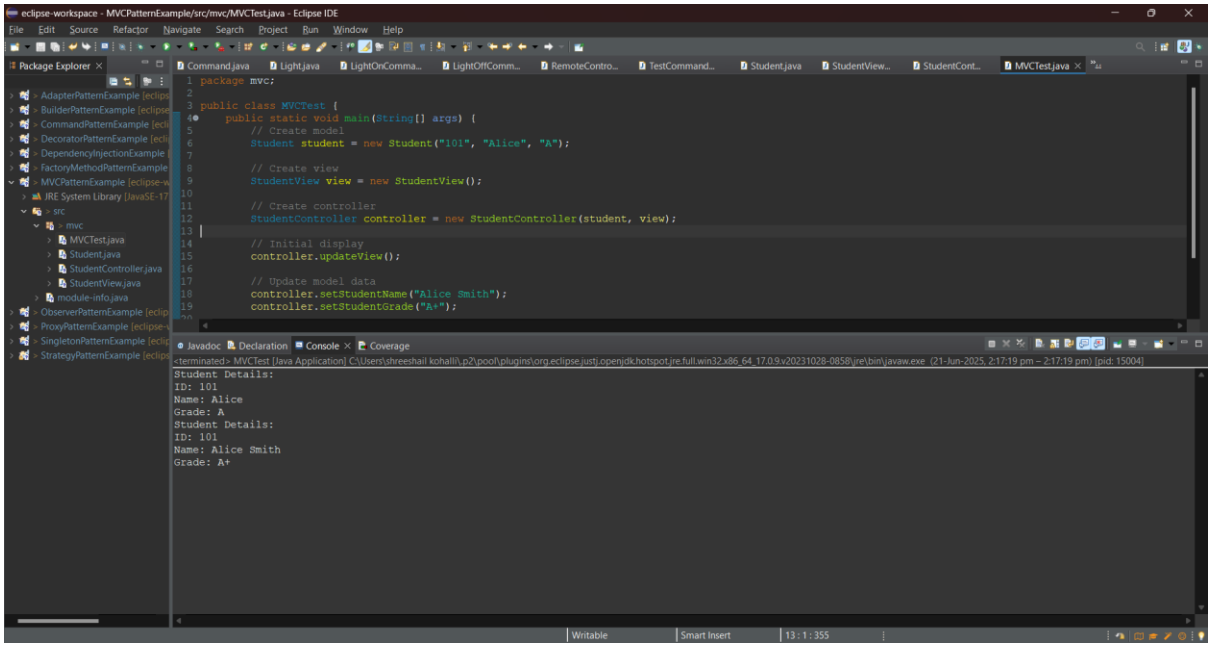
```java
package mvc;

public 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);
    }
}
```

```java
package mvc;

public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
```

```
        this.model = model;
        this.view = view;
    }

    // Controller methods to manipulate model
    public void setStudentName(String name) {
        model.setName(name);
    }

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

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

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

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

```
package mvc;

public class MVCTest {
    public static void main(String[] args) {
        // Create model
        Student student = new Student("101", "Alice", "A");

        // Create view
        StudentView view = new StudentView();

        // Create controller
        StudentController controller = new StudentController(student,
view);

        // Initial display
        controller.updateView();

        // Update model data
        controller.setStudentName("Alice Smith");
        controller.setStudentGrade("A+");

        // Display updated data
        controller.updateView();
    }
}
```

OUTPUT:



Exercise 11: Implementing Dependency Injection

```java
package di;

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

```java
package di;

public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public String findCustomerById(String customerId) {
        // Simulate database lookup
        return "Customer Name for ID " + customerId;
    }
}
```

```java
package di;

public class CustomerService {
    private final CustomerRepository customerRepository;

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

    public void printCustomer(String customerId) {
        String customer = customerRepository.findCustomerById(customerId);
        System.out.println("Customer Details: " + customer);
    }
}
```

```java
package di;

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

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

        // Use the service
        service.printCustomer("C101");
    }
}
```

OUTPUT: