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

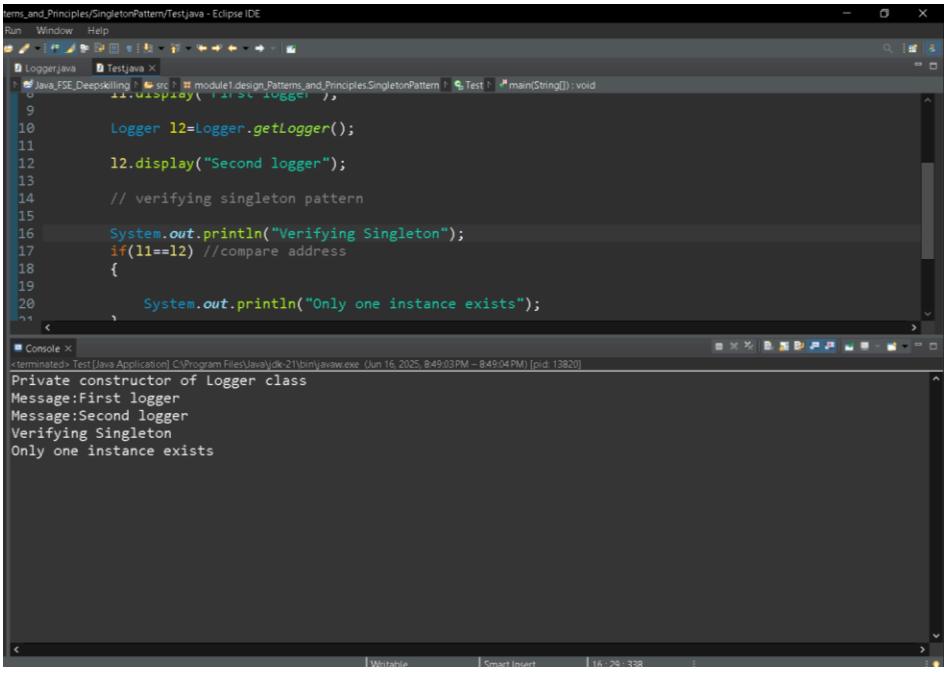
```
package module1.design_Patterns_and_Principles.SingletonPattern;

public class Logger {
	private static Logger l;
	private Logger() {
		System.out.println("Private constructor of Logger class");
	}
	//helper method
	public static Logger getLogger()
	{
		if(l==null)
		{
			l=new Logger();// lazy instantiation
		}
		return l;
	}
	// non static method to test
	public void display(String msg)
	{
		System.out.println("Message:"+msg);
	}
}
```

Test.java

```
package module1.design_Patterns_and_Principles.SingletonPattern;
```

```
public class Test {
	public static void main(String[] args) {
		Logger l1=Logger.getLogger();
		l1.display("First logger");
		Logger l2=Logger.getLogger();
		l2.display("Second logger");
		// verifying singleton pattern
		System.out.println("Verifying Singleton");
		if(l1==l2) //compare address
			System.out.println("Only one instance exists");
		else
			System.out.println("Multiple instance----Singleton not achieved");
	}
}
```

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

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;

public interface Document {
	void open(); // abstract method
}
```

WordDocument.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;

public class WordDocument implements Document {
	@Override
	public void open()
	{
	  System.out.println("Opened Word Document");
	}
}
```

PdfDocument.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;

public class PdfDocument implements Document{
	@Override
	public void open()
	{
	  System.out.println("Opened PDF Document");
	}
}
```

ExcelDocument.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public class ExcelDocument implements Document {
        @Override
        public void open()
        {
          System.out.println("Opened Excel Document");
        }
}
```

DocumentFactory.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public abstract class DocumentFactory  {
                public abstract Document createDocument();
}
```

WordFactory.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public class WordFactory extends DocumentFactory{
   @Override
   public Document createDocument()
   {
        System.out.println("Word created");
        return new WordDocument();
   }
}
```

Pdfactory.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public class PdfFactory extends DocumentFactory{
```

```
        @Override
        public Document createDocument()
    {
        System.out.println("PDF created");
        return new PdfDocument();
    }
}
```

ExcelFactory.java

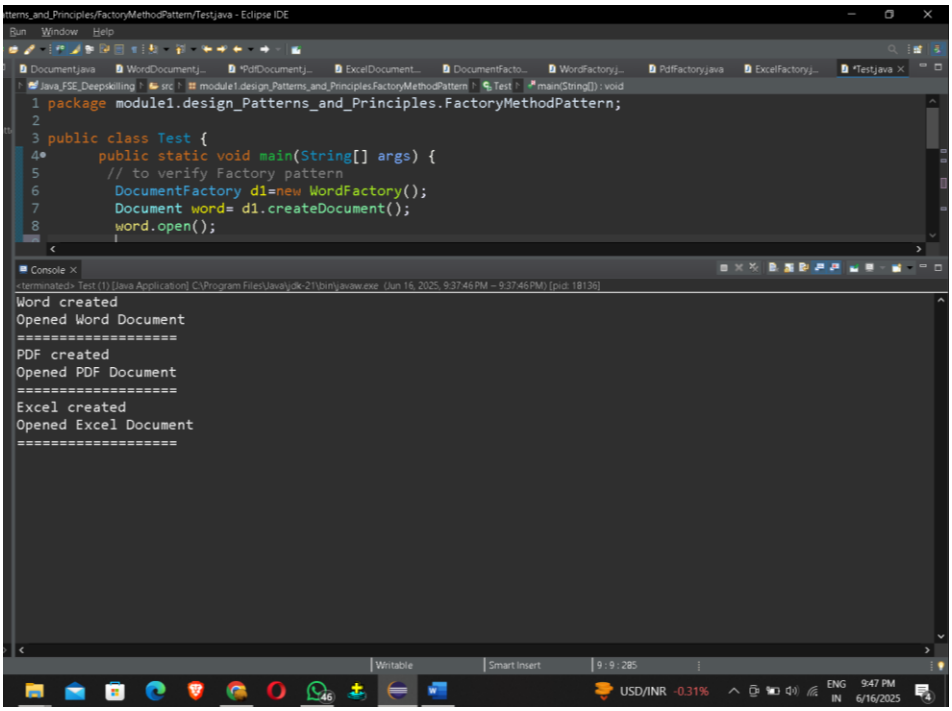
```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public class ExcelFactory extends DocumentFactory{
        @Override
        public Document createDocument()
    {
        System.out.println("Excel created");
        return new ExcelDocument();
    }
}
```

Test.java

```
package module1.design_Patterns_and_Principles.FactoryMethodPattern;
public class Test {
     public static void main(String[] args) {
          // to verify Factory pattern
        DocumentFactory d1=new WordFactory();
        Document word= d1.createDocument();
        word.open();
                System.out.println("====================");
        DocumentFactory d2=new PdfFactory();
        Document pdf= d2.createDocument();
```

```
        pdf.open();
            System.out.println("====================");
        DocumentFactory d3=new ExcelFactory();
        Document excel= d3.createDocument();
        excel.open();
            System.out.println("====================");
    }
}
```

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

package module1.design_Patterns_and_Principles.BuilderPattern;

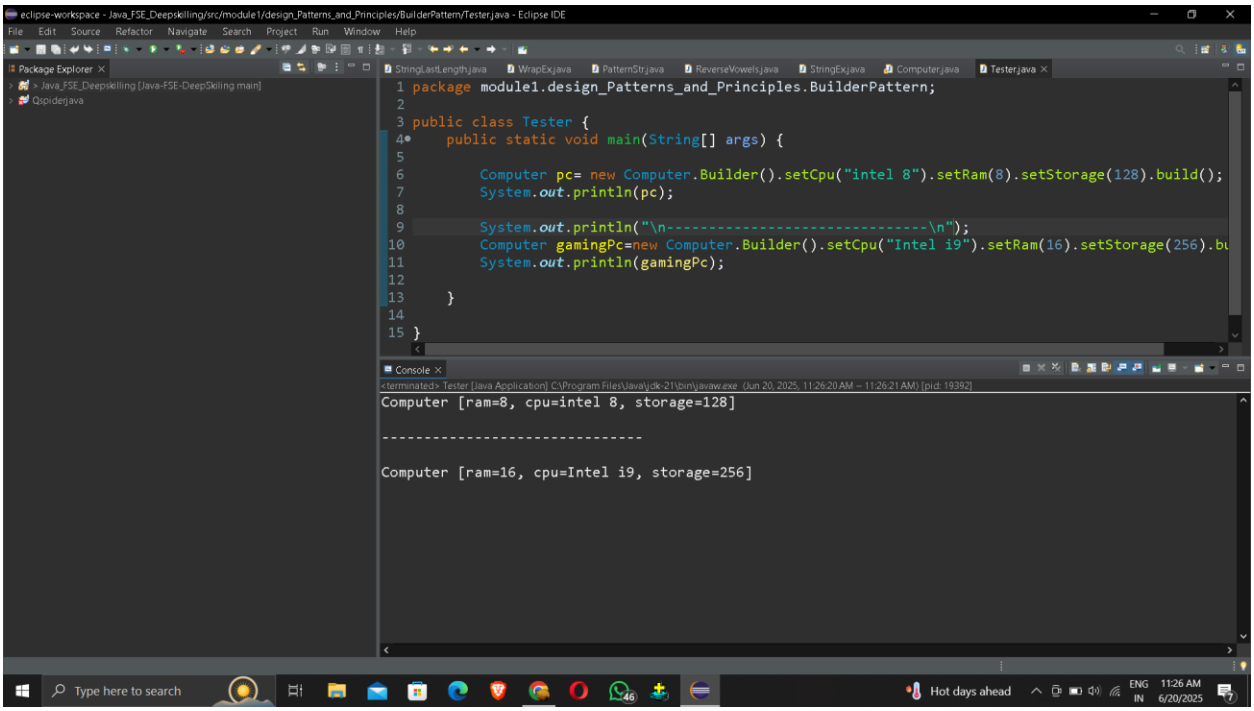
```
public class Computer {
	private int ram;
	private String cpu;
	private int storage;

	private Computer(Builder b) {
		this.ram = b.ram;
		this.cpu = b.cpu;
		this.storage = b.storage;
	}
	// to display the configurations
	@Override
	public String toString() {
		return "Computer [ram=" + ram + ", cpu=" + cpu + ", storage=" + storage + "]";
	}
	public static class Builder{
		static int ram;
		static String cpu;
		static int storage;
		public Builder setRam(int ram)
		{
			Builder.ram=ram;
			return this;
		}
		public Builder setCpu(String cpu) {
			Builder.cpu = cpu;
			return this;
		}
```

```
        public Builder setStorage(int storage) {
            Builder.storage = storage;
            return this;
        }
        public Computer build()
        {
            return new Computer(this);
        }
    }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.BuilderPattern;

public class Tester {
    public static void main(String[] args) {
        Computer pc= new Computer.Builder().setCpu("intel
8").setRam(8).setStorage(128).build();
        System.out.println(pc);
        System.out.println("\n-------------------------------\n");
        Computer gamingPc=new Computer.Builder().setCpu("Intel
i9").setRam(16).setStorage(256).build();
        System.out.println(gamingPc);
    }
}
```

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.

PaymentProcessor.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;

public interface PaymentProcessor {

    void processPayment(double amount);

}
```

GpayAdapter.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;


public class GpayAdapter implements PaymentProcessor{

                private Gpay gpay;
```

```
        public GpayAdapter(Gpay gpay) {
            this.gpay = gpay;
        }
        @Override
        public void processPayment(double amount) {
            gpay.makePayment(amount);
        }
    }
```

PayPalAdapter.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;

public class PayPalAdapter implements PaymentProcessor{


        private PayPal payPal;

        public PayPalAdapter(PayPal payPal) {
            this.payPal = payPal;
        }

        @Override
        public void processPayment(double amount) {
            payPal.sendPayment(amount);
        }
    }
```

Gpay.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;
```

```
public class PayPalAdapter implements PaymentProcessor{
        private PayPal payPal;

        public PayPalAdapter(PayPal payPal) {
            this.payPal = payPal;
        }

        @Override
        public void processPayment(double amount) {
            payPal.sendPayment(amount);
        }
    }
```

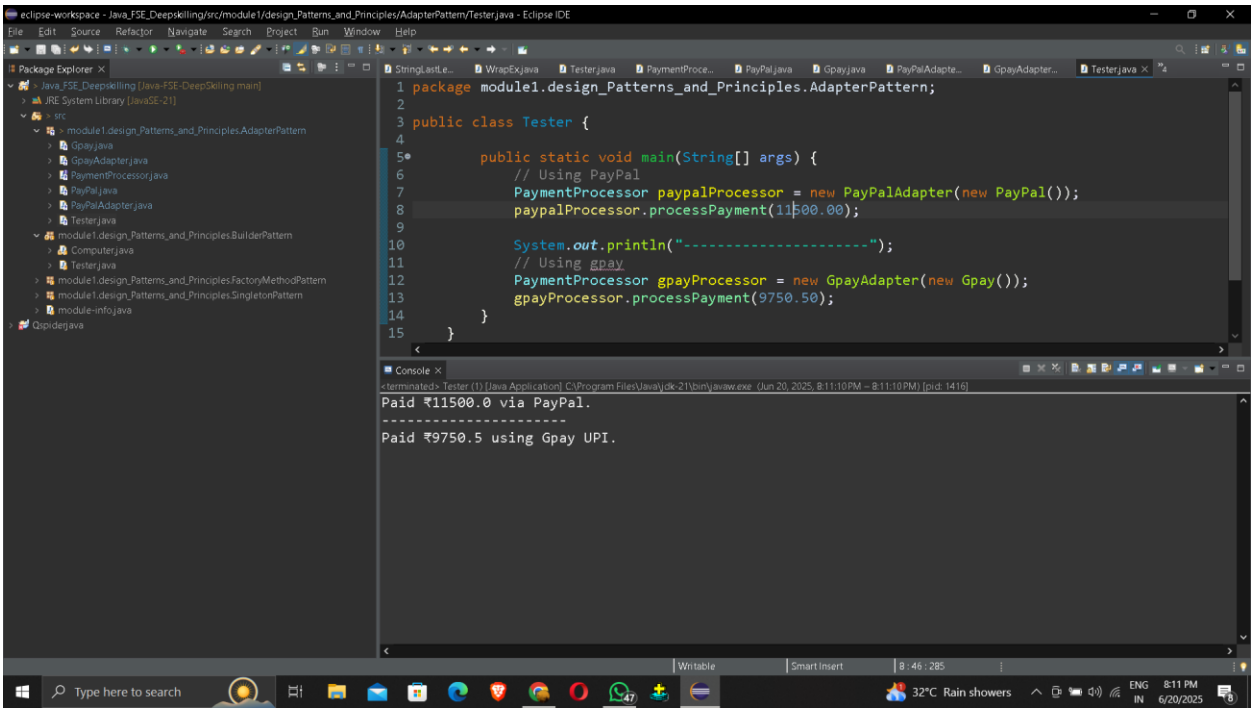
PayPal.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;

public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Paid ₹" + amount + " via PayPal.");
    }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.AdapterPattern;

public class Tester {

        public static void main(String[] args) {
            // Using PayPal
            PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPal());
```

```
        paypalProcessor.processPayment(11500.00);


        System.out.println("----------------------");
        // Using gpay
        PaymentProcessor gpayProcessor = new GpayAdapter(new Gpay());
        gpayProcessor.processPayment(9750.50);
    }
}
```



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.

Notifier.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

public interface Notifier {
```

```
    void send(String message);
}
```

NotifierDecorator.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

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);
  }
}
```

SlackNotifierDecorator.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

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("Slack message sent: " + message);
    }
}
```

SmsNotifierDecorator.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

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("SMS sent: " + message);
        }
}
```

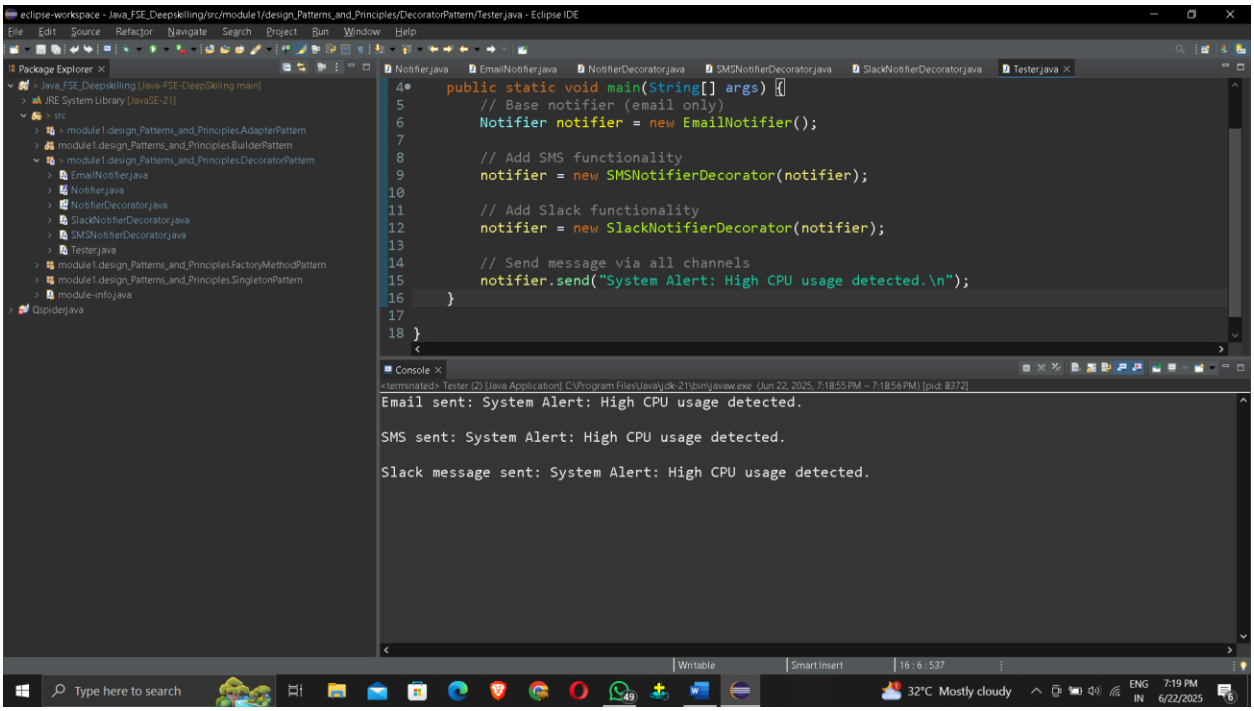
EmailNotifier.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

public class EmailNotifier implements Notifier {
```

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

}
```

Tester.java

```
package module1.design_Patterns_and_Principles.DecoratorPattern;

public class Tester {
        public static void main(String[] args) {
        // Base notifier (email only)
        Notifier notifier = new EmailNotifier();

        // Add SMS functionality
        notifier = new SMSNotifierDecorator(notifier);

        // Add Slack functionality
        notifier = new SlackNotifierDecorator(notifier);

        // Send message via all channels
        notifier.send("System Alert: High CPU usage detected.\n");
    }
}
```



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.

Image.java

```
package module1.design_Patterns_and_Principles.ProxyPattern;

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

RealImage.java

```
package module1.design_Patterns_and_Principles.ProxyPattern;


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

}
```

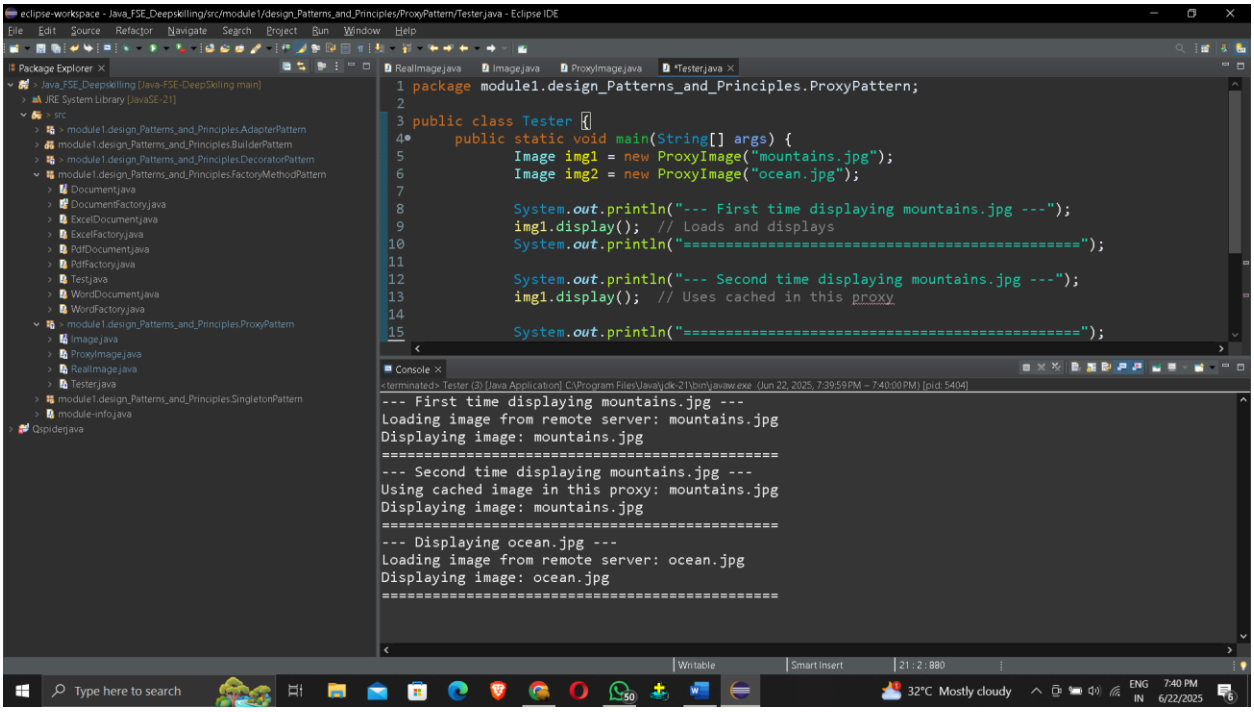
ProxyImage.java

```
package module1.design_Patterns_and_Principles.ProxyPattern;
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 initialization
```

```
        } else {
            System.out.println("Using cached image in this proxy: " + filename);
        }

        realImage.display();
    }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.ProxyPattern;
public class Tester {
        public static void main(String[] args) {
            Image img1 = new ProxyImage("mountains.jpg");
            Image img2 = new ProxyImage("ocean.jpg");

            System.out.println("--- First time displaying mountains.jpg ---");
            img1.display();  // Loads and displays
        System.out.println("================================================");

            System.out.println("--- Second time displaying mountains.jpg ---");
            img1.display();  // Uses cached in this proxy

        System.out.println("================================================");
            System.out.println("--- Displaying ocean.jpg ---");
            img2.display();  // Loads and displays

System.out.println("================================================");
}
}
```



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.

Observer.java

```
package module1.design_Patterns_and_Principles.ObserverPattern;

public interface Observer {
        void update(double price);
}
```

Stock.java

```
package module1.design_Patterns_and_Principles.ObserverPattern;

public interface Stock {
        void registerObserver(Observer observer);
    void removeObserver(Observer observer);
```

```
    void notifyObservers();
}
```

StockMarket.java

```
package module1.design_Patterns_and_Principles.ObserverPattern;
        import java.util.ArrayList;
        import java.util.List;
        public class StockMarket implements Stock {
            private List<Observer> observers = new ArrayList<>();
            private double stockPrice;
            @Override
            public void registerObserver(Observer observer) {
                observers.add(observer);
            }
            @Override
            public void removeObserver(Observer observer) {
                observers.remove(observer);
            }
            @Override
            public void notifyObservers() {
                for (Observer obs : observers) {
                    obs.update(stockPrice);
                }
            }

            public void setStockPrice(double price) {
                this.stockPrice = price;
                notifyObservers();
            }
```

```
        }
```

MobileApp.java

```
package module1.design_Patterns_and_Principles.ObserverPattern;

public class MobileApp implements Observer{
        private String name;

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

          @Override
          public void update(double price) {
             System.out.println("MobileApp " + name + ": Stock price updated to ₹" + price);
          }
}
```

WebApp.java

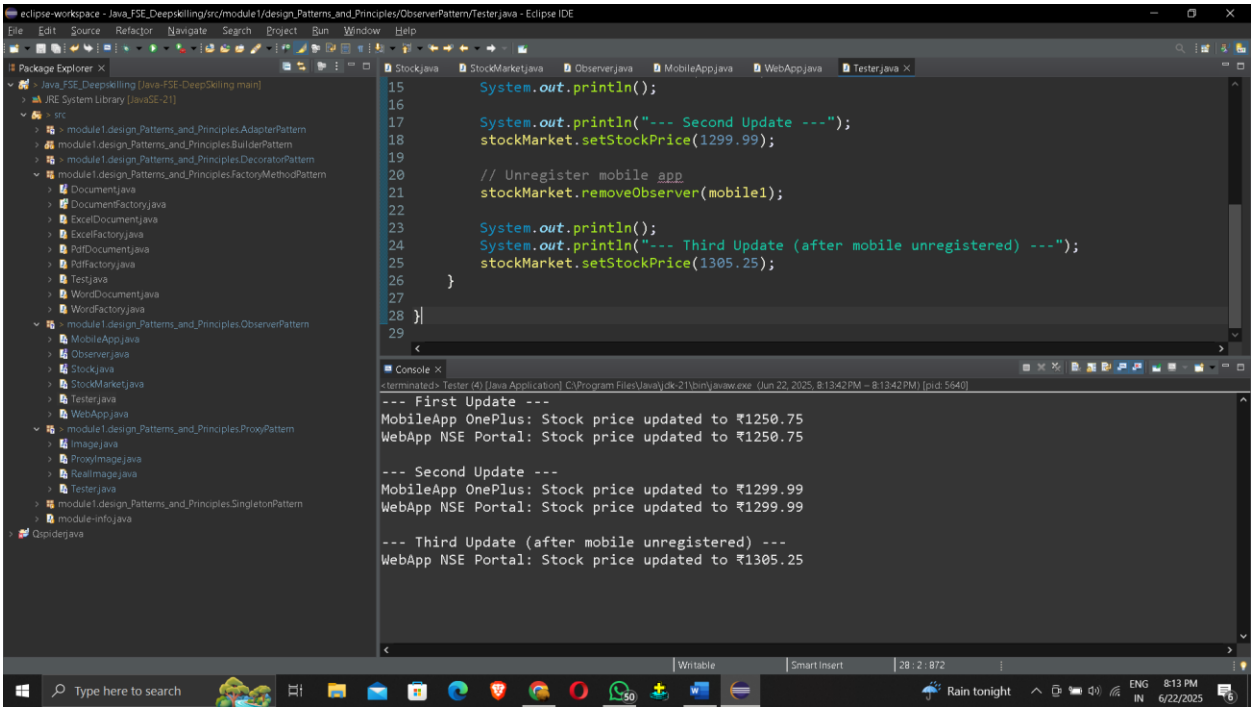
```
package module1.design_Patterns_and_Principles.ObserverPattern;

public class WebApp implements Observer{
        private String name;

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

   @Override
```

```
    public void update(double price) {
        System.out.println("WebApp " + name + ": Stock price updated to ₹" + price);
    }

}
```

Tester.java

```
package module1.design_Patterns_and_Principles.ObserverPattern;
public class Tester {
        public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobile1 = new MobileApp("OnePlus");
        Observer web1 = new WebApp("NSE Portal");

        stockMarket.registerObserver(mobile1);
        stockMarket.registerObserver(web1);

        System.out.println("--- First Update ---");
        stockMarket.setStockPrice(1250.75);
        System.out.println();

        System.out.println("--- Second Update ---");
        stockMarket.setStockPrice(1299.99);

        // Unregister mobile app
        stockMarket.removeObserver(mobile1);
        System.out.println();
        System.out.println("--- Third Update (after mobile unregistered) ---");
```

```
        stockMarket.setStockPrice(1305.25);
    }
}
```



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.

PaymentStrategy.java

```
package module1.design_Patterns_and_Principles.StrategyPattern;

public interface PaymentStrategy {

        void pay(double amount);


}
```

CreditCardPayment.java

```
package module1.design_Patterns_and_Principles.StrategyPattern;
```

```
public class CreditCardPayment implements PaymentStrategy {
        private String cardNumber;
    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }
    @Override
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);
    }
}
```

UpiPayment.java

```
package module1.design_Patterns_and_Principles.StrategyPattern;

public class UpiPayment implements PaymentStrategy{

        private String email;

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

    @Override
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using UPI: " + email);
    }
}
```

PayemntContext.java

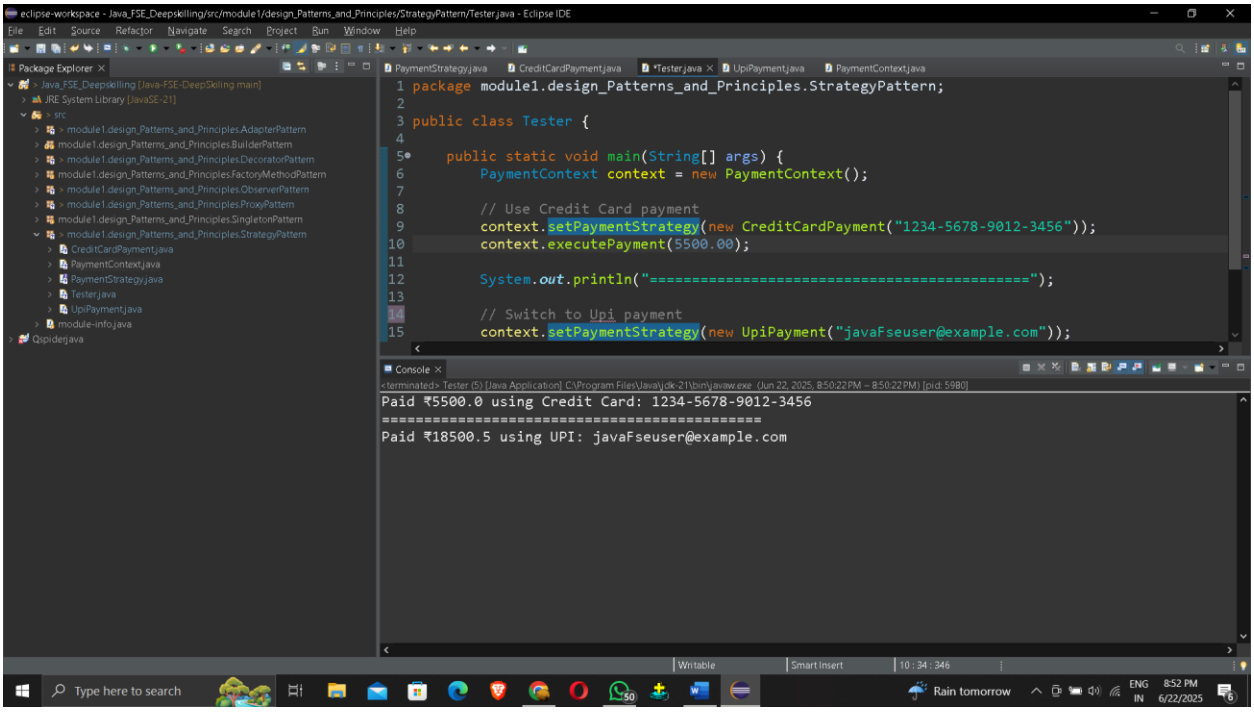
```
package module1.design_Patterns_and_Principles.StrategyPattern;
```

```
public class PaymentContext {
        private PaymentStrategy strategy;

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

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

Tester.java

```
package module1.design_Patterns_and_Principles.StrategyPattern;

public class Tester {

        public static void main(String[] args) {
      PaymentContext context = new PaymentContext();
      // Use Credit Card payment
      context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));
      context.executePayment(5500.00);
```

```
        System.out.println("==========================================");

        // Switch to Upi payment
        context.setPaymentStrategy(new UpiPayment("javaFseuser@example.com"));
        context.executePayment(18500.50);
    }
}
```



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.

Command.java

package module1.design_Patterns_and_Principles.CommandPattern;

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

}
```

Light.java

```
package module1.design_Patterns_and_Principles.CommandPattern;
public class Light {
        public void turnOn() {
      System.out.println("Light is ON");
   }
   public void turnOff() {
      System.out.println("Light is OFF");
   }
}
```

LightOnCommand.java

```
package module1.design_Patterns_and_Principles.CommandPattern;

public class LightOnCommand implements Command {
         private Light light;
            public LightOnCommand(Light light) {
               this.light = light;
            }
            @Override
            public void execute() {
               light.turnOn();
            }
}
```

LightOffCommand.java

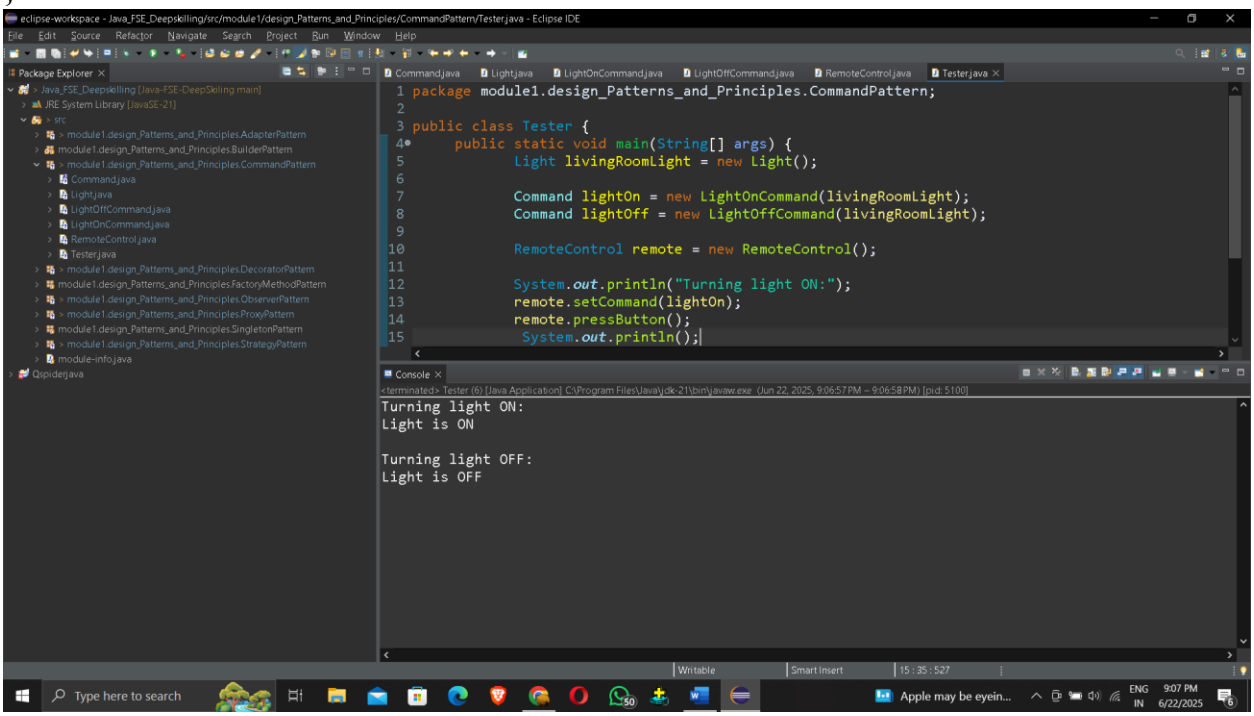
```
package module1.design_Patterns_and_Principles.CommandPattern;
public class LightOffCommand implements Command {
        private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
}
```

RemoteControl.java

```
package module1.design_Patterns_and_Principles.CommandPattern;
public class RemoteControl {
        private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println("No command set.");
        }
    }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.CommandPattern;
public class Tester {
	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();
		System.out.println("Turning light OFF:");
		remote.setCommand(lightOff);
		remote.pressButton();
	  }
}
```



Exercise 10: Implementing the MVC Pattern

Scenario:

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

Student.java

```
package module1.design_Patterns_and_Principles.MVCPattern;

public class Student {
	private String name;
   private String id;
   private String grade;
   // Constructor
   public Student(String name, String id, String grade) {
      this.name = name;
      this.id = id;
      this.grade = grade;
   }
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
```

```
        }
        public String getGrade() {
                return grade;
        }

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

    // Getters and Setters

}
```

StudentView.java

```
package module1.design_Patterns_and_Principles.MVCPattern;

public 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);
    }
}
```

StudentController.java

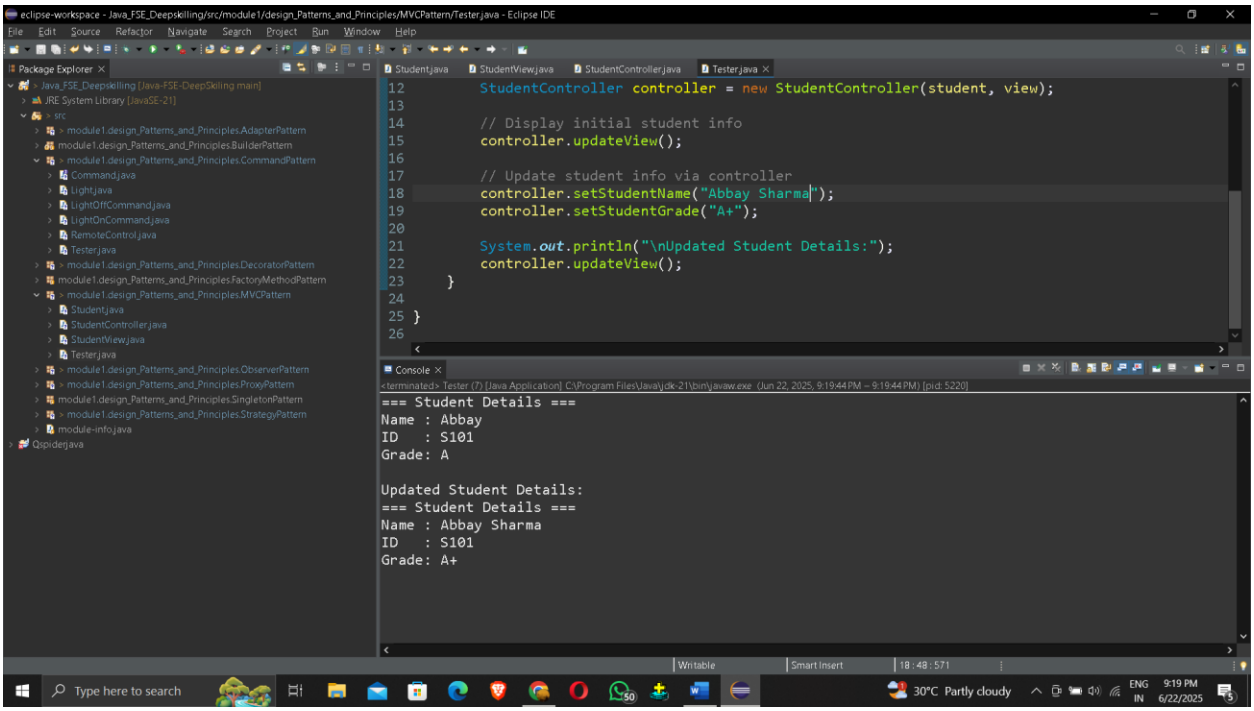
```
package module1.design_Patterns_and_Principles.MVCPattern;
public class StudentController {
```

```
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 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());
        }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.MVCPattern;
public class Tester {
        public static void main(String[] args) {
    // Create model
    Student student = new Student("Abbay", "S101", "A");

    // Create view
    StudentView view = new StudentView();
    // Create controller
    StudentController controller = new StudentController(student, view);
    // Display initial student info
    controller.updateView();
    // Update student info via controller
    controller.setStudentName("Abbay Sharma");
    controller.setStudentGrade("A+");

    System.out.println("\nUpdated Student Details:");
    controller.updateView();
  }

}
```



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.

CustomerRepository.java

```
package module1.design_Patterns_and_Principles.DependencyInversionPrinciple;

public interface CustomerRepository {

        String findCustomerById(String id);

}
```

CustomerRepositoryImpl.java

```
package module1.design_Patterns_and_Principles.DependencyInversionPrinciple;


public class CustomerRepositoryImpl implements CustomerRepository {

   @Override

   public String findCustomerById(String id) {
```

```
        // Dummy data
        return "Customer[ID: " + id + ", Name: Hinata]";
    }
}
```

CustomerService.java

```
package module1.design_Patterns_and_Principles.DependencyInversionPrinciple;

public class CustomerService {
            private CustomerRepository repository;
    // Constructor Injection
    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }
    public void showCustomerDetails(String id) {
        String customer = repository.findCustomerById(id);
        System.out.println("Customer Found: " + customer);
    }
}
```

Tester.java

```
package module1.design_Patterns_and_Principles.DependencyInversionPrinciple;

public class Tester {

            public static void main(String[] args) {
        // Create repository implementation
        CustomerRepository repo = new CustomerRepositoryImpl();
```

```
        // Inject dependency into service using constructor
        CustomerService service = new CustomerService(repo);

        // Use the service
        service.showCustomerDetails("C1008791");
    }
}
```