**Cognizant Deep Nurture 4.0 Hands-on Exercise    
   
   
   
   
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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

   
1.Define the Computer Product Class

// File: Computer.java

public class Computer {

// Required parameters

private final String CPU;

private final String RAM;

// Optional parameters

private final String storage;

private final String graphicsCard;

private final String operatingSystem;

// Private constructor that takes a Builder

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

// Static nested Builder class

public static class Builder {

private final String CPU;

private final String RAM;

private String storage;

private String graphicsCard;

private String operatingSystem;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer Configuration:\n" +

"CPU: " + CPU + "\n" +

"RAM: " + RAM + "\n" +

"Storage: " + (storage != null ? storage : "Not specified") + "\n" +

"Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not specified") + "\n" +

"Operating System: " + (operatingSystem != null ? operatingSystem : "Not specified");

}

}

2. Test the Builder Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Basic configuration

Computer basicComputer = new Computer.Builder("Intel i3", "4GB")

.build();

// Advanced configuration

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setOperatingSystem("Windows 11 Pro")

.build();

// Display configurations

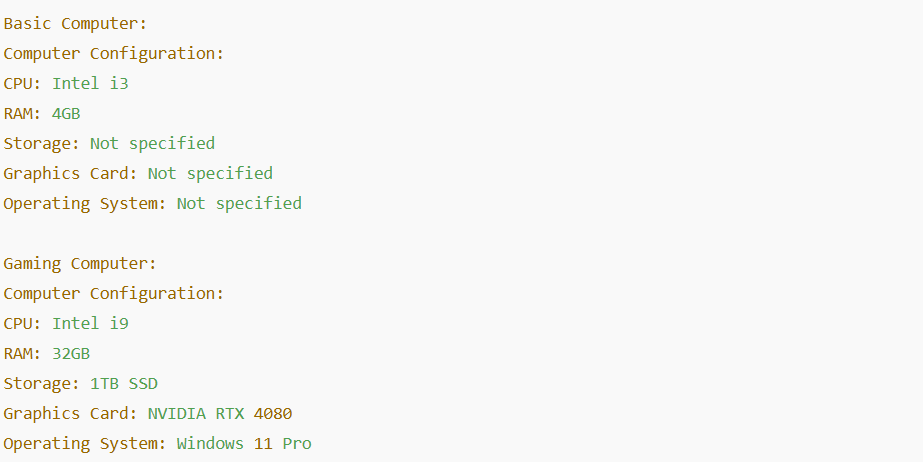
System.out.println("Basic Computer:\n" + basicComputer);

System.out.println("\nGaming Computer:\n" + gamingComputer);

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

1.Define the Target Interface

// File: PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

2. Implement Adaptee Classes (3rd Party Gateways)

// File: StripePayment.java

public class StripePayment {

public void makeStripePayment(double amount) {

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

}

}

// File: PayPalPayment.java

public class PayPalPayment {

public void sendPayPalPayment(double amount) {

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

}

}

3.Implement Adapter Classes

// File: StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripePayment stripe;

public StripeAdapter(StripePayment stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

// File: PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalPayment paypal;

public PayPalAdapter(PayPalPayment paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.sendPayPalPayment(amount);

}

}

4. Test the Adapter Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Using Stripe through adapter

PaymentProcessor stripeProcessor = new StripeAdapter(new StripePayment());

stripeProcessor.processPayment(100.00);

// Using PayPal through adapter

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalPayment());

paypalProcessor.processPayment(200.00);

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

1.Define Component Interface

// File: Notifier.java

public interface Notifier {

void send(String message);

}

2. Implement Concrete Component

// File: EmailNotifier.java

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

3. Implement Decorator Classes

Abstract Decorator

// File: NotifierDecorator.java

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);

}

}

Concrete Decorators

// File: SMSNotifierDecorator.java

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

java

CopyEdit

// File: SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

4.Test the Decorator Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Create base notifier

Notifier emailNotifier = new EmailNotifier();

// Decorate with SMS

Notifier emailAndSMSNotifier = new SMSNotifierDecorator(emailNotifier);

// Decorate with Slack

Notifier allNotifier = new SlackNotifierDecorator(emailAndSMSNotifier);

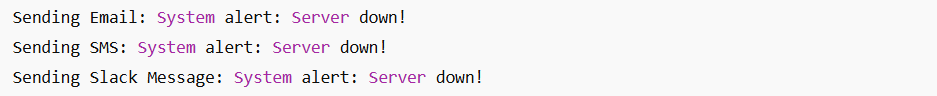
// Send notification

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

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

   
   
1.Define Subject Interface

// File: Image.java

public interface Image {

void display();

}

2. Implement Real Subject Class

// File: RealImage.java

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);

}

}

3.Implement Proxy Class

// File: ProxyImage.java

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

} else {

System.out.println("Using cached image for: " + filename);

}

realImage.display();

}

}

4.Test the Proxy Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

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

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

// First time: image is loaded

image1.display();

// Second time: image is cached

image1.display();

// First load of second image

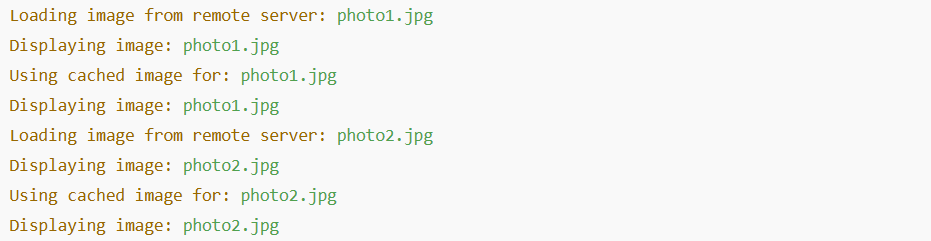
image2.display();

// Cached access

image2.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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

1.Define Subject Interface

// File: Stock.java

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

2. Implement Concrete Subject

// File: StockMarket.java

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 observer : observers) {

observer.update(stockPrice);

}

}

public void setStockPrice(double newPrice) {

System.out.println("\nStock price updated to: $" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

3.Define Observer Interface

// File: Observer.java

public interface Observer {

void update(double newPrice);

}

4.Implement Concrete Observers

// File: MobileApp.java

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(double newPrice) {

System.out.println("[" + appName + "] - New stock price: $" + newPrice);

}

}

// File: WebApp.java

public class WebApp implements Observer {

private String websiteName;

public WebApp(String websiteName) {

this.websiteName = websiteName;

}

@Override

public void update(double newPrice) {

System.out.println("[" + websiteName + "] - Stock price updated to: $" + newPrice);

}

}

5.Test the Observer Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Create subject

StockMarket stockMarket = new StockMarket();

// Create observers

Observer mobileApp = new MobileApp("StockTracker Mobile");

Observer webApp = new WebApp("StockWatch Web");

// Register observers

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

// Change stock price

stockMarket.setStockPrice(150.25);

stockMarket.setStockPrice(153.75);

// Remove an observer and change price again

stockMarket.removeObserver(mobileApp);

stockMarket.setStockPrice(158.60);

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

1.Define Strategy Interface

// File: PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

2. Implement Concrete Strategies

// File: CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

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

}

}

// File: PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account [" + email + "]");

}

}

3.Implement Context Class

// File: PaymentContext.java

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

if (paymentStrategy == null) {

System.out.println("Payment method not selected.");

} else {

paymentStrategy.pay(amount);

}

}

}

4. Test the Strategy Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Paying with Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432", "Alice Johnson"));

context.executePayment(250.00);

// Paying with PayPal

context.setPaymentStrategy(new PayPalPayment("alice@example.com"));

context.executePayment(125.50);

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

1.Define Command Interface

// File: Command.java

public interface Command {

void execute();

}

2. Implement Concrete Commands

// File: LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// File: LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

3. Implement Invoker Class

// File: RemoteControl.java

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.");

}

}

}

4.Implement Receiver Class

// File: Light.java

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

5.Test the Command Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Receiver

Light livingRoomLight = new Light();

// Commands

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Invoker

RemoteControl remote = new RemoteControl();

// Execute ON

remote.setCommand(lightOn);

remote.pressButton();

// Execute OFF

remote.setCommand(lightOff);

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**

Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

1.Define Model Class

// File: Student.java

public class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and setters

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;

}

}

2. Define View Class

// File: StudentView.java

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);

}

}

3. Define Controller Class

// File: StudentController.java

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Methods to update 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);

}

// Method to update the view

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

4. Test the MVC Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Create Model

Student student = new Student("John Doe", "S123", "A");

// Create View

StudentView view = new StudentView();

// Create Controller

StudentController controller = new StudentController(student, view);

// Initial view

controller.updateView();

// Update model via controller

controller.setStudentName("Jane Smith");

controller.setStudentGrade("A+");

// View updated model

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**

Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

1.Define Repository Interface

// File: CustomerRepository.java

public interface CustomerRepository {

String findCustomerById(String id);

}

2. Implement Concrete Repository

// File: CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

// Simulated database lookup

return "Customer [ID: " + id + ", Name: Alice Johnson]";

}

}

3. Define Service Class

// File: CustomerService.java

public class CustomerService {

private final CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

String customer = customerRepository.findCustomerById(id);

System.out.println(customer);

}

}

4. Test the Dependency Injection Implementation

// File: Main.java

public class Main {

public static void main(String[] args) {

// Create repository (dependency)

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject dependency into service

CustomerService service = new CustomerService(repository);

// Use the service

service.getCustomerDetails("C101");

}

}

OUTPUT:

760b44b0c51489489a9f4e1a33479c3d.png