**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named SingletonPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name SingletonPatternExample.
4. Click Finish.

**Step 2: Define a Singleton Class**

**Create a class named Logger:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter Logger as the class name.
4. Click Finish.

**Implement the Logger class:**

public class Logger {

// Private static instance of the class

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {

}

// Public static method to provide access to the instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Example method to log messages

public void log(String message) {

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

}

}

**Step 3: Implement the Singleton Pattern**

The above implementation ensures that the Logger class follows the Singleton design pattern by:

* Having a private static instance of itself.
* A private constructor to prevent instantiation from other classes.
* A public static method to provide access to the instance.

**Step 4: Test the Singleton Implementation**

**Create a test class to verify the Singleton implementation:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter SingletonTest as the class name.
4. Click Finish.

**Implement the SingletonTest class:**

public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

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

if (logger1 == logger2) {

System.out.println("logger1 and logger2 are the same instance.");

} else {

System.out.println("logger1 and logger2 are different instances.");

}

}

}

**Explanation:**

* In Logger class:
  + The instance is a private static member, ensuring that there is only one instance.
  + The constructor is private to prevent instantiation from outside the class.
  + The getInstance method returns the single instance, creating it if it doesn't exist.
* In SingletonTest class:
  + We obtain two references (logger1 and logger2) to the Logger instance.
  + We log messages using both references.
  + We verify if both references point to the same instance.

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named FactoryMethodPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name FactoryMethodPatternExample.
4. Click Finish.

**Step 2: Define Document Classes**

**Create interfaces or abstract classes for different document types:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Document as the interface name.
4. Click Finish.

public interface Document {

void open();

void save();

void close();

}

**Step 3: Create Concrete Document Classes**

**Implement concrete classes for each document type:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter WordDocument as the class name.
4. Click Finish.

public class WordDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving Word document...");

}

@Override

public void close() {

System.out.println("Closing Word document...");

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PdfDocument as the class name.
4. Click Finish.

public class PdfDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving PDF document...");

}

@Override

public void close() {

System.out.println("Closing PDF document...");

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter ExcelDocument as the class name.
4. Click Finish.

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving Excel document...");

}

@Override

public void close() {

System.out.println("Closing Excel document...");

}

}

**Step 4: Implement the Factory Method**

**Create an abstract class DocumentFactory:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter DocumentFactory as the class name.
4. Click Finish.

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**Create concrete factory classes for each document type:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter WordDocumentFactory as the class name.
4. Click Finish.

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PdfDocumentFactory as the class name.
4. Click Finish.

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter ExcelDocumentFactory as the class name.
4. Click Finish.

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**Step 5: Test the Factory Method Implementation**

**Create a test class to demonstrate the creation of different document types:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter FactoryMethodTest as the class name.
4. Click Finish.

public class FactoryMethodTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDocument = wordFactory.createDocument();

wordDocument.open();

wordDocument.save();

wordDocument.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDocument = pdfFactory.createDocument();

pdfDocument.open();

pdfDocument.save();

pdfDocument.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDocument = excelFactory.createDocument();

excelDocument.open();

excelDocument.save();

excelDocument.close();

}

}

**Explanation:**

* **Document Interface**: Defines the common methods that all document types should implement.
* **Concrete Document Classes**: WordDocument, PdfDocument, and ExcelDocument implement the Document interface.
* **DocumentFactory**: An abstract class that declares the createDocument method.
* **Concrete Factory Classes**: WordDocumentFactory, PdfDocumentFactory, and ExcelDocumentFactory extend DocumentFactory and implement the createDocument method to return an instance of the respective document type.
* **FactoryMethodTest**: Demonstrates the usage of the factory method pattern by creating different document types and calling their methods.

**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.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named BuilderPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name BuilderPatternExample.
4. Click Finish.

**Step 2: Define a Product Class**

**Create a class named Computer with attributes like CPU, RAM, Storage, etc.:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter Computer as the class name.
4. Click Finish.

**Implement the Computer class:**

public class Computer {

// Required attributes

private final String CPU;

private final int RAM;

// Optional attributes

private final int storage;

private final boolean isSSD;

private final boolean hasGraphicsCard;

// Private constructor to be used by the Builder class

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.isSSD = builder.isSSD;

this.hasGraphicsCard = builder.hasGraphicsCard;

}

@Override

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + "GB, Storage=" + storage + "GB, SSD=" + isSSD + ", GraphicsCard=" + hasGraphicsCard + "]";

}

// Static nested Builder class

public static class Builder {

// Required attributes

private final String CPU;

private final int RAM;

// Optional attributes

private int storage = 0;

private boolean isSSD = false;

private boolean hasGraphicsCard = false;

// Builder constructor with required attributes

public Builder(String CPU, int RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

// Setter methods for optional attributes

public Builder setStorage(int storage) {

this.storage = storage;

return this;

}

public Builder setSSD(boolean isSSD) {

this.isSSD = isSSD;

return this;

}

public Builder setGraphicsCard(boolean hasGraphicsCard) {

this.hasGraphicsCard = hasGraphicsCard;

return this;

}

// Build method to create a Computer instance

public Computer build() {

return new Computer(this);

}

}

}

**Step 3: Implement the Builder Pattern**

* **Private Constructor**: The Computer class has a private constructor that takes the Builder as a parameter.
* **Builder Class**: The static nested Builder class has methods to set optional attributes and a build() method to return a Computer instance.

**Step 4: Test the Builder Implementation**

**Create a test class to demonstrate the creation of different configurations of Computer:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter BuilderPatternTest as the class name.
4. Click Finish.

**Implement the BuilderPatternTest class:**

public class BuilderPatternTest {

public static void main(String[] args) {

// Create a Computer instance with only required attributes

Computer basicComputer = new Computer.Builder("Intel i5", 8).build();

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

// Create a Computer instance with additional optional attributes

Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", 16)

.setStorage(512)

.setSSD(true)

.setGraphicsCard(true)

.build();

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

// Create another configuration of Computer

Computer officeComputer = new Computer.Builder("Intel i7", 16)

.setStorage(256)

.setSSD(true)

.build();

System.out.println("Office Computer: " + officeComputer);

}

}

**Explanation:**

* **Product Class (Computer)**:
  + Has both required and optional attributes.
  + The private constructor initializes the Computer object using the Builder.
  + toString() method provides a readable representation of the Computer object.
* **Builder Class**:
  + Contains methods to set optional attributes and returns an instance of Computer through the build() method.
* **Test Class (BuilderPatternTest)**:
  + Demonstrates how to create different configurations of the Computer object using the Builder class.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named AdapterPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name AdapterPatternExample.
4. Click Finish.

**Step 2: Define Target Interface**

**Create an interface PaymentProcessor:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter PaymentProcessor as the interface name.
4. Click Finish.

**Implement the PaymentProcessor interface:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Step 3: Implement Adaptee Classes**

**Create classes for different payment gateways:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PayPalGateway as the class name.
4. Click Finish.

**Implement the PayPalGateway class:**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal.");

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StripeGateway as the class name.
4. Click Finish.

**Implement the StripeGateway class:**

public class StripeGateway {

public void charge(double amount) {

System.out.println("Charging $" + amount + " through Stripe.");

}

}

**Step 4: Implement the Adapter Class**

**Create adapter classes for each payment gateway:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PayPalAdapter as the class name.
4. Click Finish.

**Implement the PayPalAdapter class:**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.makePayment(amount);

}

}

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StripeAdapter as the class name.
4. Click Finish.

**Implement the StripeAdapter class:**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.charge(amount);

}

}

**Step 5: Test the Adapter Implementation**

**Create a test class to demonstrate the use of different payment gateways through the adapter:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter AdapterPatternTest as the class name.
4. Click Finish.

**Implement the AdapterPatternTest class:**

public class AdapterPatternTest {

public static void main(String[] args) {

// Create PayPal gateway and its adapter

PayPalGateway payPalGateway = new PayPalGateway();

PaymentProcessor payPalProcessor = new PayPalAdapter(payPalGateway);

// Create Stripe gateway and its adapter

StripeGateway stripeGateway = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripeGateway);

// Process payments using different gateways

payPalProcessor.processPayment(150.00);

stripeProcessor.processPayment(200.00);

}

}

**Explanation:**

* **Target Interface (PaymentProcessor)**:
  + Defines the method processPayment that will be used by the client code.
* **Adaptee Classes**:
  + PayPalGateway and StripeGateway are third-party payment gateways with different methods.
* **Adapter Classes**:
  + PayPalAdapter and StripeAdapter implement the PaymentProcessor interface and translate the calls to the methods of their respective gateways.
* **Test Class (AdapterPatternTest)**:
  + Demonstrates how to use the adapters to interact with different payment gateways using a common interface.

**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.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named DecoratorPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name DecoratorPatternExample.
4. Click Finish.

**Step 2: Define Component Interface**

**Create an interface Notifier:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Notifier as the interface name.
4. Click Finish.

**Implement the Notifier interface:**

public interface Notifier {

void send(String message);

}

**Step 3: Implement Concrete Component**

**Create a class EmailNotifier that implements Notifier:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter EmailNotifier as the class name.
4. Click Finish.

**Implement the EmailNotifier class:**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**Step 4: Implement Decorator Classes**

**Create an abstract decorator class NotifierDecorator:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter NotifierDecorator as the class name.
4. Click Finish.

**Implement the NotifierDecorator class:**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

**Create a concrete decorator class SMSNotifierDecorator:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter SMSNotifierDecorator as the class name.
4. Click Finish.

**Implement the SMSNotifierDecorator class:**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // send via the base notifier

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

**Create a concrete decorator class SlackNotifierDecorator:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter SlackNotifierDecorator as the class name.
4. Click Finish.

**Implement the SlackNotifierDecorator class:**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // send via the base notifier

sendSlackMessage(message);

}

private void sendSlackMessage(String message) {

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

}

}

**Step 5: Test the Decorator Implementation**

**Create a test class to demonstrate sending notifications via multiple channels:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter DecoratorPatternTest as the class name.
4. Click Finish.

**Implement the DecoratorPatternTest class:**

public class DecoratorPatternTest {

public static void main(String[] args) {

// Create a base EmailNotifier

Notifier emailNotifier = new EmailNotifier();

// Decorate EmailNotifier with SMSNotifierDecorator

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

// Further decorate with SlackNotifierDecorator

Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

// Send message through all channels

slackNotifier.send("Hello, this is a test message!");

}

}

**Explanation:**

* **Component Interface (Notifier)**:
  + Defines the method send that will be used by the client code.
* **Concrete Component (EmailNotifier)**:
  + Implements the Notifier interface and sends an email.
* **Decorator Class (NotifierDecorator)**:
  + Implements the Notifier interface and holds a reference to another Notifier object.
  + Delegates the send call to the wrapped notifier.
* **Concrete Decorators (SMSNotifierDecorator, SlackNotifierDecorator)**:
  + Extend NotifierDecorator and add functionality to send SMS and Slack messages respectively.
* **Test Class (DecoratorPatternTest)**:
  + Demonstrates how to use decorators to enhance the base EmailNotifier with additional functionalities.

**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.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named ProxyPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name ProxyPatternExample.
4. Click Finish.

**Step 2: Define Subject Interface**

**Create an interface Image:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Image as the interface name.
4. Click Finish.

**Implement the Image interface:**

public interface Image {

void display();

}

**Step 3: Implement Real Subject Class**

**Create a class RealImage that implements Image:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter RealImage as the class name.
4. Click Finish.

**Implement the RealImage class:**

public class RealImage implements Image {

private String imageFileName;

public RealImage(String imageFileName) {

this.imageFileName = imageFileName;

loadImageFromServer();

}

private void loadImageFromServer() {

System.out.println("Loading image: " + imageFileName);

}

@Override

public void display() {

System.out.println("Displaying image: " + imageFileName);

}

}

**Step 4: Implement Proxy Class**

**Create a class ProxyImage that implements Image:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter ProxyImage as the class name.
4. Click Finish.

**Implement the ProxyImage class:**

public class ProxyImage implements Image {

private RealImage realImage;

private String imageFileName;

private boolean isLoaded = false;

public ProxyImage(String imageFileName) {

this.imageFileName = imageFileName;

}

@Override

public void display() {

if (!isLoaded) {

realImage = new RealImage(imageFileName);

isLoaded = true;

}

realImage.display();

}

}

**Step 5: Test the Proxy Implementation**

**Create a test class to demonstrate the use of ProxyImage:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter ProxyPatternTest as the class name.
4. Click Finish.

**Implement the ProxyPatternTest class:**

public class ProxyPatternTest {

public static void main(String[] args) {

// Create a ProxyImage object

Image image1 = new ProxyImage("test\_image1.jpg");

// Display image using proxy

image1.display(); // This will load the image from server and display it

image1.display(); // This will display the cached image

// Create another ProxyImage object

Image image2 = new ProxyImage("test\_image2.jpg");

// Display image using proxy

image2.display(); // This will load the image from server and display it

}

}

**Explanation:**

* **Subject Interface (Image)**:
  + Defines the method display that will be used by both the RealImage and ProxyImage classes.
* **Real Subject Class (RealImage)**:
  + Implements the Image interface and simulates loading an image from a remote server. The constructor triggers the image loading process.
* **Proxy Class (ProxyImage)**:
  + Implements the Image interface and manages the RealImage instance. It supports lazy initialization (loading the image only when it's needed) and caching (storing the loaded image to avoid reloading).
* **Test Class (ProxyPatternTest)**:
  + Demonstrates how to use ProxyImage to load and display images. The image is only loaded once and reused from the cache on subsequent calls.

**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.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named ObserverPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name ObserverPatternExample.
4. Click Finish.

**Step 2: Define Subject Interface**

**Create an interface Stock:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Stock as the interface name.
4. Click Finish.

**Implement the Stock interface:**

java

Copy code

import java.util.List;

public interface Stock {

void registerObserver(Observer observer);

void deregisterObserver(Observer observer);

void notifyObservers();

}

**Step 3: Implement Concrete Subject**

**Create a class StockMarket that implements Stock:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StockMarket as the class name.
4. Click Finish.

**Implement the StockMarket class:**

java

Copy code

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private String stockName;

private double stockPrice;

public StockMarket(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

this.observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void deregisterObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

}

**Step 4: Define Observer Interface**

**Create an interface Observer:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Observer as the interface name.
4. Click Finish.

**Implement the Observer interface:**

java

Copy code

public interface Observer {

void update(String stockName, double stockPrice);

}

**Step 5: Implement Concrete Observers**

**Create a class MobileApp that implements Observer:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter MobileApp as the class name.
4. Click Finish.

**Implement the MobileApp class:**

java

Copy code

public class MobileApp implements Observer {

@Override

public void update(String stockName, double stockPrice) {

System.out.println("MobileApp: Stock " + stockName + " has a new price: " + stockPrice);

}

}

**Create a class WebApp that implements Observer:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter WebApp as the class name.
4. Click Finish.

**Implement the WebApp class:**

java

Copy code

public class WebApp implements Observer {

@Override

public void update(String stockName, double stockPrice) {

System.out.println("WebApp: Stock " + stockName + " has a new price: " + stockPrice);

}

}

**Step 6: Test the Observer Implementation**

**Create a test class to demonstrate the registration and notification of observers:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter ObserverPatternTest as the class name.
4. Click Finish.

**Implement the ObserverPatternTest class:**

java

Copy code

public class ObserverPatternTest {

public static void main(String[] args) {

// Create the stock market

StockMarket stockMarket = new StockMarket("TechCorp", 150.00);

// Create observers

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

// Register observers

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

// Update stock price

System.out.println("Updating stock price...");

stockMarket.setStockPrice(155.00);

// Remove an observer and update stock price again

stockMarket.deregisterObserver(mobileApp);

System.out.println("Updating stock price...");

stockMarket.setStockPrice(160.00);

}

}

**Explanation:**

* **Subject Interface (Stock)**:
  + Defines methods for registering, deregistering, and notifying observers.
* **Concrete Subject Class (StockMarket)**:
  + Implements the Stock interface and maintains a list of observers. It notifies all registered observers when the stock price changes.
* **Observer Interface (Observer)**:
  + Defines the update method that observers will use to receive notifications.
* **Concrete Observers (MobileApp, WebApp)**:
  + Implement the Observer interface and provide specific behaviors for handling notifications.
* **Test Class (ObserverPatternTest)**:
  + Demonstrates how to use the StockMarket and Observer implementations. It registers observers, updates stock prices, and shows how observers are notified of changes.

**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.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named StrategyPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name StrategyPatternExample.
4. Click Finish.

**Step 2: Define Strategy Interface**

**Create an interface PaymentStrategy:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter PaymentStrategy as the interface name.
4. Click Finish.

**Implement the PaymentStrategy interface:**

java

Copy code

public interface PaymentStrategy {

void pay(double amount);

}

**Step 3: Implement Concrete Strategies**

**Create a class CreditCardPayment that implements PaymentStrategy:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter CreditCardPayment as the class name.
4. Click Finish.

**Implement the CreditCardPayment class:**

java

Copy code

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

private String cvv;

private String expirationDate;

public CreditCardPayment(String cardNumber, String cardHolderName, String cvv, String expirationDate) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

this.cvv = cvv;

this.expirationDate = expirationDate;

}

@Override

public void pay(double amount) {

System.out.println(amount + " paid with credit card.");

}

}

**Create a class PayPalPayment that implements PaymentStrategy:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PayPalPayment as the class name.
4. Click Finish.

**Implement the PayPalPayment class:**

java

Copy code

public class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

@Override

public void pay(double amount) {

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

}

}

**Step 4: Implement Context Class**

**Create a class PaymentContext:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter PaymentContext as the class name.
4. Click Finish.

**Implement the PaymentContext class:**

java

Copy code

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(double amount) {

paymentStrategy.pay(amount);

}

}

**Step 5: Test the Strategy Implementation**

**Create a test class to demonstrate selecting and using different payment strategies:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StrategyPatternTest as the class name.
4. Click Finish.

**Implement the StrategyPatternTest class:**

java

Copy code

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using credit card

PaymentStrategy creditCardPayment = new CreditCardPayment("1234567890123456", "John Doe", "123", "12/23");

context.setPaymentStrategy(creditCardPayment);

context.pay(250.75);

// Pay using PayPal

PaymentStrategy payPalPayment = new PayPalPayment("john.doe@example.com", "password123");

context.setPaymentStrategy(payPalPayment);

context.pay(250.75);

}

}

**Explanation:**

* **Strategy Interface (PaymentStrategy)**:
  + Defines the pay method that all payment strategies must implement.
* **Concrete Strategies (CreditCardPayment, PayPalPayment)**:
  + Implement the PaymentStrategy interface and provide specific behaviors for handling payments.
* **Context Class (PaymentContext)**:
  + Holds a reference to a PaymentStrategy and uses it to execute the payment.
* **Test Class (StrategyPatternTest)**:
  + Demonstrates how to use different payment strategies by setting them in the context and calling the pay method.

**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**.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named CommandPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name CommandPatternExample.
4. Click Finish.

**Step 2: Define Command Interface**

**Create an interface Command:**

1. Right-click on the src folder in your project.
2. Select New > Interface.
3. Enter Command as the interface name.
4. Click Finish.

**Implement the Command interface:**

java

Copy code

public interface Command {

void execute();

}

**Step 3: Implement Concrete Commands**

**Create a class LightOnCommand that implements Command:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter LightOnCommand as the class name.
4. Click Finish.

**Implement the LightOnCommand class:**

java

Copy code

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**Create a class LightOffCommand that implements Command:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter LightOffCommand as the class name.
4. Click Finish.

**Implement the LightOffCommand class:**

java

Copy code

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**Step 4: Implement Invoker Class**

**Create a class RemoteControl:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter RemoteControl as the class name.
4. Click Finish.

**Implement the RemoteControl class:**

java

Copy code

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**Step 5: Implement Receiver Class**

**Create a class Light:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter Light as the class name.
4. Click Finish.

**Implement the Light class:**

java

Copy code

public class Light {

public void turnOn() {

System.out.println("The light is on.");

}

public void turnOff() {

System.out.println("The light is off.");

}

}

**Step 6: Test the Command Implementation**

**Create a test class to demonstrate issuing commands using the RemoteControl:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter CommandPatternTest as the class name.
4. Click Finish.

**Implement the CommandPatternTest class:**

java

Copy code

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn on the light

remote.setCommand(lightOn);

remote.pressButton();

// Turn off the light

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Explanation:**

* **Command Interface (Command)**:
  + Defines the execute method that all commands must implement.
* **Concrete Commands (LightOnCommand, LightOffCommand)**:
  + Implement the Command interface and provide specific behaviors for turning the light on and off.
* **Invoker Class (RemoteControl)**:
  + Holds a reference to a Command and uses it to execute the command when the button is pressed.
* **Receiver Class (Light)**:
  + Contains the actual operations to turn the light on and off.
* **Test Class (CommandPatternTest)**:
  + Demonstrates how to issue commands using the remote control.

**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**.

Solution:

**Step 1: Create a New Java Project**

**Create a new Java project named MVCPatternExample:**

1. Open Eclipse.
2. Select File > New > Java Project.
3. Enter the project name MVCPatternExample.
4. Click Finish.

**Step 2: Define Model Class**

**Create a class Student:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter Student as the class name.
4. Click Finish.

**Implement the Student class:**

java

Copy code

public class Student {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = 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;

}

}

**Step 3: Define View Class**

**Create a class StudentView:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StudentView as the class name.
4. Click Finish.

**Implement the StudentView class:**

java

Copy code

public class StudentView {

public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

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

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

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

System.out.println("Grade: " + studentGrade);

}

}

**Step 4: Define Controller Class**

**Create a class StudentController:**

1. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter StudentController as the class name.
4. Click Finish.

**Implement the StudentController class:**

java

Copy code

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**Step 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. Right-click on the src folder in your project.
2. Select New > Class.
3. Enter MVCPatternTest as the class name.
4. Click Finish.

**Implement the MVCPatternTest class:**

java

Copy code

public class MVCPatternTest {

public static void main(String[] args) {

// Create the model

Student model = new Student("1", "John Doe", "A");

// Create the view

StudentView view = new StudentView();

// Create the controller

StudentController controller = new StudentController(model, view);

// Update and display the student details

controller.updateView();

// Update the student name

controller.setStudentName("Jane Doe");

// Update and display the student details

controller.updateView();

}

}

**Explanation:**

* **Model Class (Student)**:
  + Represents the data structure for a student with attributes like id, name, and grade.
* **View Class (StudentView)**:
  + Provides a method displayStudentDetails to display student details.
* **Controller Class (StudentController)**:
  + Acts as an intermediary between the model and view, handling the logic for updating the model and refreshing the view.
* **Test Class (MVCPatternTest)**:
  + Demonstrates creating a Student object, updating its details using the StudentController, and displaying the updated details using the StudentView.