# Week 3. Structural Design Patterns

1. **Adapting Different Payment Gateways**

**Objective**: Integrate multiple payment gateways (e.g., PayPal, Stripe) into a single payment processing interface.

* 1. Create Interfaces: Define a PaymentProcessor interface with methods like processPayment(amount: double).
  2. Implement Concrete Adapters: Implement adapters for each payment gateway (e.g., PayPalAdapter, StripeAdapter) that adapt their specific APIs to the PaymentProcessor interface.
  3. Test Integration: Create a client class that uses the PaymentProcessor interface to process payments, and test with different adapters.
  4. Analyze the issue in this scenario that the Adapter pattern is meant to address.

**Code : Class PaymentClient**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalAPI {

void pay(double amount) {

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

}

}

class StripeAPI {

void makePayment(double amount) {

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

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalAPI payPalAPI;

PayPalAdapter(PayPalAPI payPalAPI) {

this.payPalAPI = payPalAPI;

}

@Override

public void processPayment(double amount) {

payPalAPI.pay(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeAPI stripeAPI;

StripeAdapter(StripeAPI stripeAPI) {

this.stripeAPI = stripeAPI;

}

@Override

public void processPayment(double amount) {

stripeAPI.makePayment(amount);

}

}

public class PaymentClient {

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter(new PayPalAPI());

PaymentProcessor stripe = new StripeAdapter(new StripeAPI());

paypal.processPayment(100.00);

stripe.processPayment(200.00);

}

}

**Output:**

A black screen with white text

Description automatically generated

# Extending Functionality of Text Processing

**Objective:** Enhance a basic text processing class with additional features (e.g., spell checking, text formatting).

* 1. Create Base Component: Define a TextProcessor class with a method process (text: String).
  2. Create Decorators: Implement decorators like SpellCheckDecorator, TextFormatDecorator that extend TextProcessor and add new functionalities.
  3. Test Decorators: Create a client class that uses decorated TextProcessor objects to process text with added functionalities.
  4. Analyze the issue in this scenario that the Decorator pattern is meant to address.

**Code : Class TextProcessor**

public class TextProcessor {

public String process(String text) {

return text;

}

public static void main(String[] args) {

String text = "This is teh example text that might recieve some errors.";

TextProcessor textProcessor = new TextProcessor();

TextProcessor spellCheckedProcessor = new SpellCheckDecorator(textProcessor);

TextProcessor formattedProcessor = new TextFormatDecorator(spellCheckedProcessor);

String result = formattedProcessor.process(text);

System.out.println(result);

}

public static class TextProcessorDecorator extends TextProcessor {

protected TextProcessor wrapped;

public TextProcessorDecorator(TextProcessor wrapped) {

this.wrapped = wrapped;

}

@Override

public String process(String text) {

return wrapped.process(text);

}

}

public static class SpellCheckDecorator extends TextProcessorDecorator {

public SpellCheckDecorator(TextProcessor wrapped) {

super(wrapped);

}

@Override

public String process(String text) {

String correctedText = text.replace("teh", "the").replace("recieve", "receive");

return super.process(correctedText);

}

}

public static class TextFormatDecorator extends TextProcessorDecorator {

public TextFormatDecorator(TextProcessor wrapped) {

super(wrapped);

}

@Override

public String process(String text) {

String formattedText = text.toUpperCase();

return super.process(formattedText);

}

}

}

**Output**



1. You are tasked with designing a video player application that supports playing different types of video formats (e.g., MP4, AVI, MKV) on different operating systems (e.g., Windows, macOS, Linux).

Using the **Bridge Design Pattern**, implement the following:

* 1. Create a common interface for video formats.
  2. Implement concrete classes for each video format (e.g., MP4Player, AVIPlayer).
  3. Create a common interface for operating systems.
  4. Implement concrete classes for each operating system (e.g., WindowsOS, LinuxOS).
  5. Demonstrate how a video player can play any video format on any operating system by bridging the two hierarchies.

# Requirements:

1. The implementation should demonstrate the decoupling of abstraction and implementation using the Bridge Design Pattern.
2. Provide a code snippet showing how a specific video format is played on a particular operating system.

**Code : Class BridgePatternDemo**

interface VideoFormat {

void play();

}

class MP4Player implements VideoFormat {

@Override

public void play() {

System.out.println("Playing MP4 video format");

}

}

class AVIPlayer implements VideoFormat {

@Override

public void play() {

System.out.println("Playing AVI video format");

}

}

interface OperatingSystem {

void run(VideoFormat videoPlayer);

}

class WindowsOS implements OperatingSystem {

@Override

public void run(VideoFormat videoPlayer) {

System.out.println("Running on Windows OS");

videoPlayer.play();

}

}

class LinuxOS implements OperatingSystem {

@Override

public void run(VideoFormat videoPlayer) {

System.out.println("Running on Linux OS");

videoPlayer.play();

}

}

class VideoPlayer {

private VideoFormat videoFormat;

private OperatingSystem os;

public VideoPlayer(VideoFormat videoFormat, OperatingSystem os) {

this.videoFormat = videoFormat;

this.os = os;

}

public void playVideo() {

os.run(videoFormat);

}

}

public class BridgePatternDemo {

public static void main(String[] args) {

VideoFormat mp4Player = new MP4Player();

VideoFormat aviPlayer = new AVIPlayer();

OperatingSystem windowsOS = new WindowsOS();

OperatingSystem linuxOS = new LinuxOS();

System.out.println("Playing MP4 on Windows OS:");

VideoPlayer mp4VideoPlayer = new VideoPlayer(mp4Player, windowsOS);

mp4VideoPlayer.playVideo();

System.out.println("\nPlaying AVI on Linux OS:");

VideoPlayer aviVideoPlayer = new VideoPlayer(aviPlayer, linuxOS);

aviVideoPlayer.playVideo();

}

}

**Output**

A screenshot of a computer

Description automatically generated

1. You are developing a financial application that tracks stock prices. Your system fetches live stock data from two different APIs:
   1. **StockAPI\_A** provides data in the format:
      1. public class StockAPIA {
      2. public double getPrice(String stockSymbol) {
      3. // Returns stock price
      4. }
      5. }
   2. **StockAPI\_B** provides data in a different format:
      1. public class StockAPIB {
      2. public String fetchStockPrice(String stockSymbol) {
      3. // Returns stock price as a String
      4. }
      5. }

Your application should work with **both APIs** seamlessly and display the stock price in a unified format.

# Requirements:

1. Implement an **Adapter Design Pattern** to unify the two APIs.
2. Create an interface StockPriceProvider with a method double getStockPrice(String stockSymbol).
3. Implement two adapters (StockAPIAAdapter and StockAPIBAdapter) to integrate StockAPI\_A and StockAPI\_B respectively.
4. Write a client class StockPriceViewer that accepts a StockPriceProvider and displays the stock price.

**Code : Class StockPriceApp**

public class StockPriceApp {

public interface StockPriceProvider {

double getStockPrice(String stockSymbol);

}

public static class StockAPIA {

public double getPrice(String stockSymbol) {

if (stockSymbol.equals("AAPL")) return 150.00;

else if (stockSymbol.equals("GOOGL")) return 2800.00;

return 0.0;

}

}

public static class StockAPIB {

public String fetchStockPrice(String stockSymbol) {

if (stockSymbol.equals("AAPL")) return "150.00";

else if (stockSymbol.equals("GOOGL")) return "2800.00";

return "0.0";

}

}

public static class StockAPIAAdapter implements StockPriceProvider {

private StockAPIA stockAPIA;

public StockAPIAAdapter(StockAPIA stockAPIA) {

this.stockAPIA = stockAPIA;

}

@Override

public double getStockPrice(String stockSymbol) {

return stockAPIA.getPrice(stockSymbol);

}

}

public static class StockAPIBAdapter implements StockPriceProvider {

private StockAPIB stockAPIB;

public StockAPIBAdapter(StockAPIB stockAPIB) {

this.stockAPIB = stockAPIB;

}

@Override

public double getStockPrice(String stockSymbol) {

String priceString = stockAPIB.fetchStockPrice(stockSymbol);

return Double.parseDouble(priceString);

}

}

public static class StockPriceViewer {

private StockPriceProvider stockPriceProvider;

public StockPriceViewer(StockPriceProvider stockPriceProvider) {

this.stockPriceProvider = stockPriceProvider;

}

public void displayStockPrice(String stockSymbol) {

double price = stockPriceProvider.getStockPrice(stockSymbol);

System.out.println("The stock price of " + stockSymbol + " is: " + price);

}

}

public static void main(String[] args) {

StockAPIA stockAPIA = new StockAPIA();

StockAPIB stockAPIB = new StockAPIB();

StockPriceProvider stockAPIAProvider = new StockAPIAAdapter(stockAPIA);

StockPriceProvider stockAPIBProvider = new StockAPIBAdapter(stockAPIB);

StockPriceViewer viewer = new StockPriceViewer(stockAPIAProvider);

viewer.displayStockPrice("AAPL");

viewer.displayStockPrice("GOOGL");

viewer = new StockPriceViewer(stockAPIBProvider);

viewer.displayStockPrice("AAPL");

viewer.displayStockPrice("GOOGL");

}

}

**Output**

A black screen with white text

Description automatically generated