Assignment 3: Refactoring and Adapter Pattern

> For this assignment, you **do not need to submit code**; it will be checked and discussed by the TA during the lab session.

Course: Object-Oriented Analysis and Design
Deadline: [Insert Date]
Points: 100

Executive Summary

In this assignment, you will step into the role of a Software Architect for an E-Commerce platform. You have inherited a legacy codebase that handles order processing. Your goal is to modernize this system in two steps:

1. **Refactor** a monolithic class to improve readability and maintainability, eliminating specific "Code Smells".
2. **Implement the Adapter Pattern** to integrate a newly acquired, incompatible payment system without breaking the existing client code.

This assignment tests your understanding of **Code Smells**, **Extract Method** refactoring, and the **Adapter Pattern**.

Scenario: The "Legacy Order System"

Your company currently uses a class named `OrderService` to handle customer orders. However, the code was poorly writtenand has degenerated under maintenance.

Additionally, the business has decided to support a new payment method, **"PayPalAPI"**, but its interface is completely different from your system's standard `PaymentGateway` interface.

Task 1: Refactoring (40 Points)

Objective

Identify "Bad Code Smells" in the provided monolithic method and apply the **Extract Method** refactoring technique to improve its design.

1.1 Analyze the Legacy Code

Below is the current implementation of the `OrderService` class. It suffers from the **Long Method** smell, where a single method does too much (printing banners, calculating totals, processing payments).

```java
// Legacy code - DO NOT USE IN PRODUCTION
public class OrderService {
    private String _customerName;
    private double _totalAmount;

    public OrderService(String name, double amount) {
        this._customerName = name;
        this._totalAmount = amount;
    }

    public void processOrder() {
        // 1. Print Banner
        System.out.println("**************************");
        System.out.println("***** ORDER PROCESS ******");
        System.out.println("**************************");

        // 2. Validate Inventory (Simulated logic)
        if (_totalAmount <= 0) {
            System.out.println("Error: Invalid order amount.");
            return;
        }
        System.out.println("Checking inventory for customer: " + _customerName);
        // ... simplistic inventory check logic ...

        // 3. Process Payment
        // (This logic is mixed directly inside the method)
        System.out.println("Connecting to Standard Payment Gateway...");
        System.out.println("Processing payment of $" + _totalAmount);
        System.out.println("Payment Successful.");

        // 4. Print Receipt
        System.out.println("Name: " + _customerName);
        System.out.println("Amount: " + _totalAmount);
        System.out.println("Date: " + java.time.LocalDate.now());
    }
}
```

1.2 Apply "Extract Method"

Requirements:
You must refactor `processOrder()` by breaking it down into smaller, semantic-preserving methods.

1. **Create a `printBanner()` method**: Encapsulate the banner printing logic.
2. **Create a `printReceipt()` method**: Encapsulate the printing of order details.
    - *Hint:* Reference the "Extract Method Refactoring Example" in Lecture 7 for how to handle local variables vs instance variables.
3. **Create a `processPayment()` method**: Encapsulate the payment logic.

Deliverable for 1.2:
Submit the refactored `OrderService` class. The `processOrder()` method should now look significantly cleaner, acting as a high-level orchestrator calling your new private methods.

> **Note:** "Refactoring usually entails small changes with large cumulative effects". Ensure the observable behavior remains exactly the same.

Task 2: Implement Adapter Pattern (60 Points)

Objective

Use the **Adapter Pattern** to convert the interface of a class into another interface the client expects.

2.1 The Standard Interface

Your system expects all payment services to implement the following interface:

```java
public interface PaymentGateway {
    // Returns true if successful
    void pay(double amount);
}
```

2.2 The Incompatible "Adaptee"

You have been given a third-party library class `PayPalAPI`. You cannot change this class (it is "closed" code).

```java
// The "Adaptee"
public class PayPalAPI {
    // Note: It uses Token-based logic and different method names
    public void makePayment(String token, float transactionAmount) {
        System.out.println("PayPal: Processing payment of " + transactionAmount + " with
    }

    public String getToken() {
        return "User-Token-123";
    }
}
```

Incompatibilities:

1. Target expects `pay(double)`, but PayPal uses `makePayment(String, float)`.
2. PayPal requires a `token` which is not present in the standard interface.
3. Data types differ (`double` vs `float`) - a form of **Primitive Obsession** or simple type mismatch.

2.3 Implement the Adapter

Create a class `PayPalAdapter` that implements `PaymentGateway`.

Requirements:

1. Implement `PaymentGateway`.
2. Use **Composition**: The adapter should hold a reference to `PayPalAPI`.
3. In the `pay(double amount)` method:
   - Retrieve the token from the PayPal instance.
   - Convert the `double` amount to `float` (cast).
   - Call `makePayment` on the PayPal instance.

> *Reference:* See the "TurkeyAdapter" example in Lecture 6 for structure.

2.4 Integrate into Refactored Code

Modify your **refactored** `OrderService` from Task 1 to support dependency injection for the payment gateway.

1. Add a field `PaymentGateway gateway` to `OrderService`.
2. Modify the constructor (or add a setter) to accept a `PaymentGateway`.
3. Update your extracted `processPayment()` method to call `gateway.pay(_totalAmount)` instead of the hardcoded print statements.

Task 3: The Client Application (Validation)

Create a `MainApp.java` class to demonstrate your work.

Steps:

1. **Scenario A (Standard):** Create an `OrderService` using a standard implementation of `PaymentGateway` (you can create a simple `CreditCardProcessor` class for this).
2. **Scenario B (Adapter):**
   - Create an instance of `PayPalAPI`.
   - Wrap it in your `PayPalAdapter`.
   - Create an `OrderService` passing in the *adapter*.
   - Call `processOrder()`.

Expected Output (Scenario B):

```
*************************
***** ORDER PROCESS ******
*************************
Checking inventory for customer: Alice
PayPal: Processing payment of 100.0 with token User-Token-123
Name: Alice
Amount: 100.0
Date: 2025-11-XX
```

Checklist

Required Files

1. `PaymentGateway.java` (Interface)
2. `PayPalAPI.java` (The Adaptee - provided)
3. `PayPalAdapter.java` (The Adapter)
4. `OrderService.java` (Refactored class using the Interface)
5. `MainApp.java` (Client code demonstrating both scenarios)