Software Design Patterns - Assignment 3

Student: Saqyp Dias

Group: SE-2436

Pattern: Adapter Pattern

# Introduction

The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work

together. It acts as a bridge between two incompatible systems by converting the interface of one class into another interface that the client expects.

In this implementation, we demonstrate the Adapter Pattern using a distance conversion scenario where Imperial data (miles) needs to be adapted to work with a Metric system (kilometers). The

adapter seamlessly converts between these two measurement systems without modifying the existing

classes.

# Code Implementation

## Step 1: Define the Target Interface

We define the interface that our client expects to work with - the Metric system.

java

public interface MetricTarget {

*// Returns the distance measurement in Kilometers (KM).*

double getDistanceInKilometers();

}

Purpose: This interface defines the contract for metric distance measurements. The client code expects to work with this interface.

## Step 2: Create the Incompatible Class (Adaptee)

The existing Imperial system that we need to adapt to work with our Metric interface.

java

public class ImperialData {

*// Data is stored in miles*

private final double miles;

public ImperialData(double miles) { this.miles = miles;

}

public double getDistanceInMiles() { return miles;

}

}

Purpose: This represents the existing legacy system that stores distance in miles. We cannot modify this class, but need to make it compatible with our Metric interface.

## Step 3: Implement the Adapter

The adapter class that bridges the gap between Imperial and Metric systems.

java

public class LengthAdapter implements MetricTarget {

*// The Adapter holds a reference to the Adaptee (ImperialData).*

private final ImperialData imperialData;

*// Using a named constant instead of a "magic number".*

private static final double MILES\_TO\_KM\_FACTOR = 1.60934;

public LengthAdapter(ImperialData imperialData) {

*// Initializes the link to the incompatible object*

this.imperialData = imperialData;

}

@Override

public double getDistanceInKilometers() {

*// 1. Get the incompatible data (miles)*

double miles = imperialData.getDistanceInMiles();

*// 2. Convert miles to kilometers*

double kilometers = miles \* MILES\_TO\_KM\_FACTOR;

*// 3. Return the adapted result*

return kilometers;

}

}

Purpose: This is the core of the Adapter Pattern. It implements the target interface ( )

MetricTarget

while internally using the adaptee ( ). The conversion logic is encapsulated here.

ImperialData

## Step 4: Client Implementation

The client code that demonstrates how the Adapter Pattern works seamlessly.

java

public class Main {

public static void main(String[] args) {

*// Input miles and convert to km*

Scanner sc = new Scanner(System.in); int inputMiles = sc.nextInt();

ImperialData imperialSource = new ImperialData(inputMiles);

MetricTarget adapter = new LengthAdapter(imperialSource); double distanceInKM = adapter.getDistanceInKilometers();

System.out.println("\nSource distance (Imperial): " +

imperialSource.getDistanceInMiles() + " miles"); System.out.println("Adapted distance (Metric): " +

distanceInKM + " km");

System.out.println("\nAdapter Pattern successfully demonstrated.");

}

}

Purpose: Demonstrates the pattern in action. The client works with the knowing about the underlying Imperial system conversion.interface without

MetricTarget

# Design Pattern Benefits Demonstrated

## Interface Compatibility

java

MetricTarget adapter = new LengthAdapter(imperialSource);

*// Client works with MetricTarget interface, unaware of Imperial system*

The client can work with Imperial data through the familiar Metric interface.

## Code Reusability

java

private final ImperialData imperialData; *// Reuses existing Imperial system*

No need to modify or replace the existing

ImperialData

## Separation of Concerns

class.

java

private static final double MILES\_TO\_KM\_FACTOR = 1.60934;

*// Conversion logic is encapsulated in the adapter*

Conversion logic is isolated in the adapter, not scattered throughout the codebase.

## Open/Closed Principle

Open for Extension: Can easily add more adapters (e.g.,

WeightAdapter

Closed for Modification: No changes needed to existing

, )

or client code

TemperatureAdapter

ImperialData

# Clean Code Principles Applied

## Single Responsibility Principle (SRP)

: Only handles Imperial distance storage

ImperialData

: Only handles Imperial-to-Metric conversion

LengthAdapter

: Only defines the Metric interface contract

MetricTarget

1. Dependency Inversion Principle (DIP)

java

MetricTarget adapter = new LengthAdapter(imperialSource);

*// Client depends on abstraction (MetricTarget), not concrete class*

1. Named Constants for Magic Numbers

java

private static final double MILES\_TO\_KM\_FACTOR = 1.60934;

*// Clear, maintainable constant instead of magic number*

1. Clear Method Names

java

getDistanceInKilometers() *// Self-documenting method name*

getDistanceInMiles() *// Clearly indicates unit of measurement*

# Sample Output

Input: 5

Source distance (Imperial): 5.0 miles Adapted distance (Metric): 8.0467 km

Adapter Pattern successfully demonstrated.

The output clearly shows the seamless conversion from Imperial (5 miles) to Metric (8.0467 km) through the adapter.

# GitHub Repository

Project Link: <https://github.com/aaituu/SoftDP/tree/main/SDP>\_assik3

# Conclusion

The Adapter Pattern was successfully implemented using a distance conversion example. The pattern demonstrates how to make incompatible interfaces work together without modifying existing code