Adapter Design Pattern in action

```
class Program
{
    static void Main(string[] args)
    {
        FahrenheitSensor fahrenheitSensor = new FahrenheitSensor();
        ICelsiusSensor celsiusSensor = new
FahrenheitToCelsiusAdapter(fahrenheitSensor);

        Console.WriteLine("Temperature in Celsius: " +
celsiusSensor.GetTemperatureCelsius());
    }
```

```
public class FahrenheitToCelsiusAdapter :
ICelsiusSensor
{
    private readonly FahrenheitSensor
_fahrenheitSensor;

    public FahrenheitToCelsiusAdapter(FahrenheitSensor
fahrenheitSensor)
    {
        _fahrenheitSensor = fahrenheitSensor;
    }

    public double GetTemperatureCelsius()
    {
        double tempFahrenheit =
_fahrenheitSensor.GetTemperatureFahrenheit();
        return (tempFahrenheit - 32) * 5.0 / 9.0;
    }
}
```

```
public class FahrenheitSensor {
  public double GetTemperatureFahrenheit() {
    // Simulate getting temperature in Fahrenheit
    return 98.6;
  }
}

public interface ICelsiusSensor {
  double GetTemperatureCelsius();
}
```

@lijotech

Usage Scenario

In this example, the **FahrenheitSensor** class represents the legacy system that provides temperature data in Fahrenheit. The **ICelsiusSensor** interface represents the new system that expects temperature data in Celsius. The **FahrenheitToCelsiusAdapter** class adapts the **FahrenheitSensor** to the **ICelsiusSensor** interface by converting the temperature from Fahrenheit to Celsius.

Merits:

- Reusability: Allows existing classes to work together without modifying their source code.
- Flexibility: Makes it easier to switch between different implementations.
- Separation of Concerns: Keeps the conversion logic separate from the business logic.

Demerits:

- Increased Complexity: Adds additional layers of abstraction, which can make the code harder to understand.
- Performance Overhead: May introduce a slight performance overhead due to the additional processing.