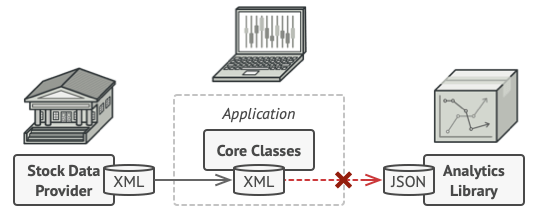
**1. Into - Need of Adapter**

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

**The Scenario:** You are building a modern **Stock Market Dashboard**.

* **The Component:** You bought a fancy 3rd-party charting library (FancyCharts) that generates beautiful graphs. It **strictly requires JSON** input.
* **The Data Source:** Your data comes from a 20-year-old legacy Stock Feed server. It **strictly returns XML**.

**The Crash:** You cannot pass the XML string into the JSON library. It will throw a syntax error immediately.



### 

namespace NeedOfAdapter

{

// 1. The Legacy System (Returns XML)

public class StockFeed

{

public string GetStockData()

{

return "<stocks><stock><symbol>AAPL</symbol><price>150</price></stock></stocks>";

}

}

// 2. The Modern Chart Library (Expects JSON)

public class FancyCharts

{

public void RenderChart(string data)

{

if(data.Trim().StartsWith("{") == false)

{

throw new Exception("Data is in JSON format. Cannot render!");

}

Console.WriteLine($"Rendering data: {data}");

}

}

// 3. The Client (Fails)

public class DashboardApp

{

public void ShowCharts()

{

StockFeed feed = new StockFeed();

FancyCharts chart = new FancyCharts();

// CRITICAL ERROR: Incompatible Formats

//chart.RenderChart(feed.GetStockData()); // THROWS EXCEPTION

}

}

public class Program

{

public static void Main(string[] args)

{

DashboardApp app = new DashboardApp();

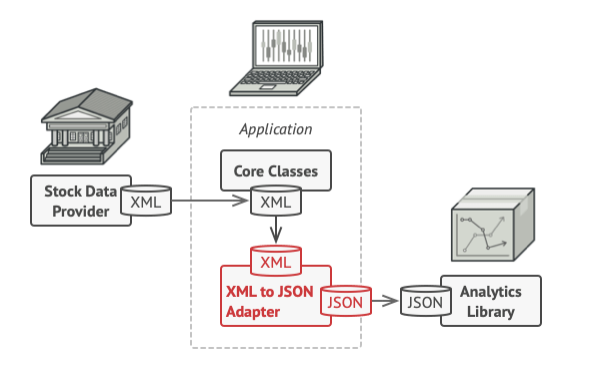
app.ShowCharts();

}

}

}

**2. Solution Using Adapter**



using Newtonsoft.Json;

using System.Xml;

namespace XMLToJson

{

// 1. The Legacy System (Returns XML)

public class StockFeed

{

public string GetStockData()

{

return "<stocks><stock><symbol>AAPL</symbol><price>150</price></stock></stocks>";

}

}

// 2. The Modern Chart Library (Expects JSON)

public class FancyCharts

{

public void RenderChart(string data)

{

if (data.Trim().StartsWith("{") == false)

{

throw new Exception("Data is in JSON format. Cannot render!");

}

Console.WriteLine($"Rendering data: {data}");

}

}

// 3. Adapter

// The Interface (What the Client expects)

// "I need a provider that gives me JSON string"

public interface IJsonDataAdapter

{

string GetJsonData();

}

public class XmlToJsonAdapter : IJsonDataAdapter

{

private readonly StockFeed \_stockFeed;

public XmlToJsonAdapter(StockFeed stockFeed)

{

\_stockFeed = stockFeed;

}

public string GetJsonData()

{

// Step A: Get the data in the original format (XML)

string xmlDocument = \_stockFeed.GetStockData();

// Step B: Convert it (The "Adaptation" logic)

XmlDocument doc = new XmlDocument();

doc.LoadXml(xmlDocument);

// Convert XML Node to JSON String

string json = JsonConvert.SerializeXmlNode(doc);

// Step C: Return the format the client expects

return json;

}

}

public class DashboardApp

{

public void ShowCharts()

{

StockFeed feed = new StockFeed();

// Wrap the legacy system in the adapter

IJsonDataAdapter adapter = new XmlToJsonAdapter(feed);

// Usage

// The App thinks it's talking to a modern JSON provider.

// It has NO IDEA that XML parsing is happening behind the scenes.

string data = adapter.GetJsonData();

Console.WriteLine($"Chart Rendered with: {data}");

}

}

// 4. The Application

public class Program

{

public static void Main(string[] args)

{

DashboardApp app = new DashboardApp();

app.ShowCharts();

}

}

}

The Adapter pattern involves a single class called the Adapter, which is responsible for joining functionalities of independent or incompatible interfaces.

**Let’s go through the components of the Adapter pattern:**

* **Target Interface:** This is the interface that the client code expects to interact with. It’s the interface that your client code uses.
* **Adaptee:** This is the class that has the functionality that the client code wants to use, but it doesn’t conform to the target interface.
* **Adapter:** This is the class that bridges the gap between the Target Interface and the Adaptee. It implements the Target Interface and delegates the calls to the Adaptee.

In this case:

**Target Interface :** FancyCharts (JSON service)

**Adaptee :** StockFeed (XML service)

**Adapter :** XMLToJson

Another Example:

//Third Party Libraries

//The Scenario: Two Incompatible SDKs

//You have two database libraries. You cannot change their code. notice they have different method names.

namespace NeedOfAdapterDatabaseExample

{

// SDK 1: MySQL

public class MySqlSdk

{

public void OpenConnection(string u, string p) { /\* ... \*/ }

public void RunQuery(string sql) { /\* ... \*/ }

}

// SDK 2: SQLite

public class SqliteSdk

{

public void Login(string dbFile) { /\* ... \*/ }

public void ExecuteCommand(string cmd) { /\* ... \*/ }

}

public class ReportService

{

private MySqlSdk \_mySqlSdk;

private SqliteSdk \_sqliteSdk;

private string \_dbType;

public ReportService(string dbType)

{

\_dbType = dbType;

if (dbType == "mysql") \_mySqlSdk = new MySqlSdk();

else if (dbType == "sqlite") \_sqliteSdk = new SqliteSdk();

}

public void GenerateReport()

{

if (\_dbType == "mysql")

{

\_mySqlSdk.OpenConnection("admin", "password");

\_mySqlSdk.RunQuery("SELECT \* FROM reports");

}

else if (\_dbType == "sqlite")

{

\_sqliteSdk.Login("data.db");

\_sqliteSdk.ExecuteCommand("SELECT \* FROM reports");

}

Console.WriteLine("Report Generated");

}

}

public class Program

{

public static void Main(string[] args)

{

ReportService reportSerivce = new("mysql");

reportSerivce.GenerateReport();

}

}

}

/\*

\* Why this is bad:-

\* Polluted Logic: The service contains unrelated "plumbing" logic (checking types, handling specific parameters).

\* Tight Coupling: The service depends directly on MySqlSdk AND SqliteSdk.

\* Violation of OCP: To add Oracle support, you must modify this class and risk breaking the reporting logic.

\*/

**Use Of Adapter in LLD**

### Scenario 1: Integrating Third-Party SDKs (The Most Common)

**The Problem:** You are building an E-commerce app. You have defined your own interface IPaymentProcessor with a method Pay(amount).

* You want to add **Stripe** support.
* You download the Stripe SDK, but their class has a method called MakeCharge(amount, currency).
* **Conflict:** You cannot change the Stripe SDK code (it's a compiled DLL/Library), and you don't want to change your core application code to match Stripe's specific naming.

**The Solution:** You create a wrapper class (StripeAdapter) that implements *your* interface (IPaymentProcessor) and translates the calls to the Stripe SDK.

### Scenario 2: Legacy Code Migration

**The Problem:** Your company uses an old legacy system for logging (OldLogger.Log(string msg)).

* Management buys a fancy new library (SplunkLogger) that offers better analytics, but it requires a strictly formatted object: Splunk.Write(EventContext ctx).
* You have 500 classes in your system calling Log("error"). You cannot rewrite all 500 classes.

**The Solution:** You create an Adapter class. The rest of the system keeps calling Log(string), but internally the Adapter converts that string into an EventContext object and calls the new library.

### Scenario 3: Data Format Translation

**The Problem:** Your system acts as a stock market dashboard.

* Your generic charts expect input in **XML** format.
* You want to integrate a new crypto-currency feed, but their API returns **JSON**.

**The Solution:** You write an adapter that accepts the JSON data, parses it, and returns the XML structure your UI expects.

**Strategy and Adapter Looks Similar**

If an interviewer asks, "Isn't this just the Strategy pattern?", the perfect answer is:

"Structurally, yes, they rely on polymorphism. But the **Adapter** is necessary here because the MySqlSdk class is **incompatible** with our IDatabase interface and we cannot modify the SDK. If we were writing the database logic from scratch to match our interface, it would be the **Strategy** pattern."

**Adapter and Strategy often look identical in code, but their *intent* is completely different.**

Here is the breakdown of why your observation is correct, but why we still call it "Adapter" in this specific scenario.

### 1. Structural Identity (Why you are right)

If you look at the UML class diagram, both patterns look like this:

* **Interface:** IDatabase (or IStrategy)
* **Concrete Classes:** MySqlAdapter (or ConcreteStrategyA)
* **Client:** ReportService (or Context)

In both cases, the Client holds an interface and calls a method, and polymorphism handles the rest. This is why you feel like it's the Strategy pattern.

### 2. The Difference is "Intent" and "Ownership"

The difference isn't in how you write the interface, but in **what problem you are solving** and **who owns the code**.

#### Case A: The Strategy Pattern

**Scenario:** You are writing a navigation app. You need to calculate a route.

* **The Problem:** "I need to support Walking, Driving, and Bicycling algorithms."
* **The Code:** You write the WalkingRoute class from scratch. You write the DrivingRoute class from scratch.
* **Key Characteristic:** You **own** all the code. You designed the classes specifically to fit your interface. There is no "mismatch" to fix.

#### Case B: The Adapter Pattern

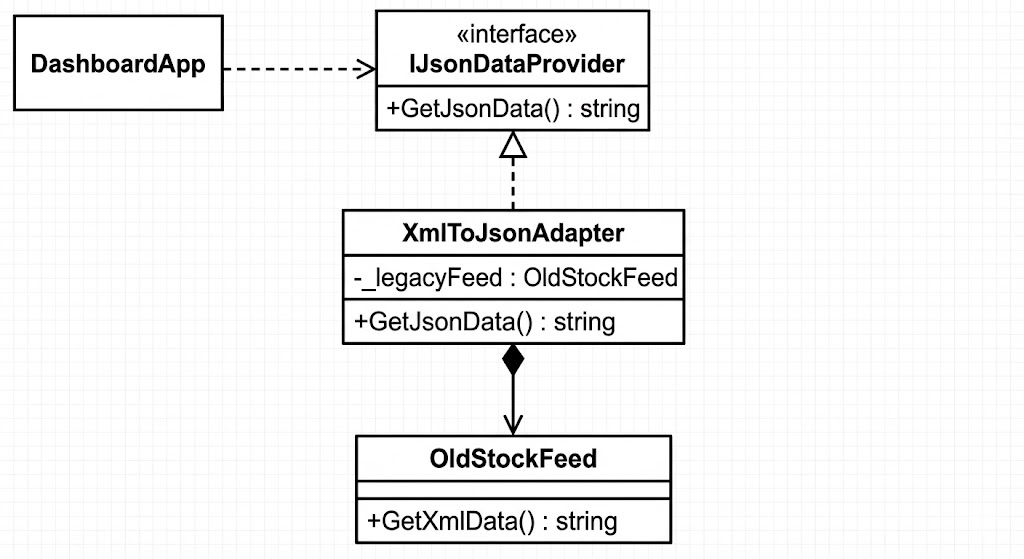
**Scenario:** You are writing the Database app we just discussed.

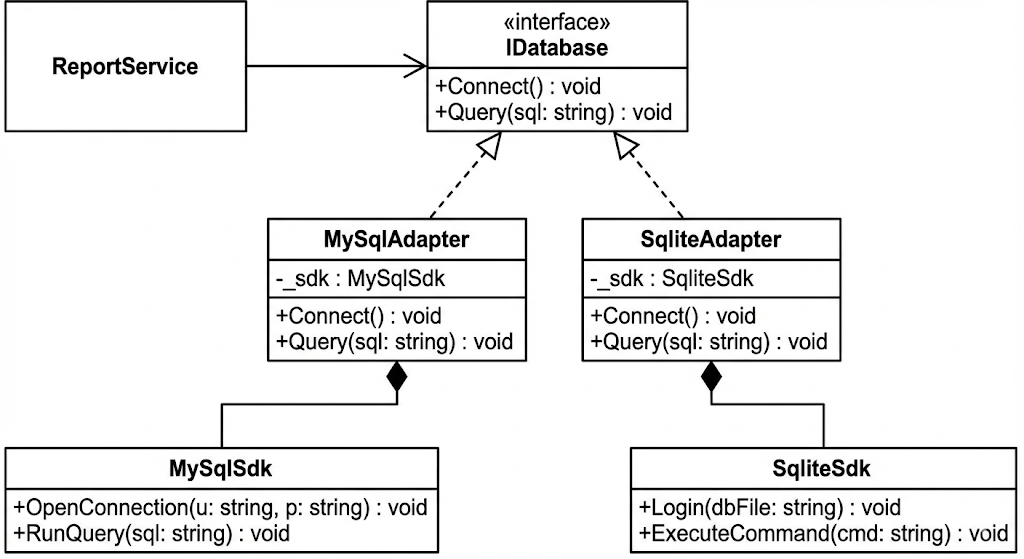
* **The Problem:** "I need to use this MySQL SDK, but its method is called OpenConnection, and my app expects Connect."
* **The Code:** You **did not** write the MySqlSdk. It is a compiled DLL from Oracle. You cannot change it.
* **Key Characteristic:** You are **fixing a mismatch**. You are writing a wrapper solely to translate "Language A" (your interface) to "Language B" (the 3rd party SDK).

### 3. Visualizing the Difference

| **Feature** | **Strategy Pattern** | **Adapter Pattern** |
| --- | --- | --- |
| **Goal** | To offer **different behaviors** (Algorithms). | To make **incompatible interfaces** work together. |
| **The "Impl"** | You write the logic inside the class. | The class is just a wrapper (pass-through) to a 3rd party. |
| **Relationship** | The Strategy **IS** the implementation. | The Adapter **HAS** the implementation (Adaptee). |
| **Analogy** | A Game Console (Change the cartridge to change the game). | A Power Plug (Travel adapter converting UK plug to US socket). |

### 





| **Pattern** | **Intent** |
| --- | --- |
| **Adapter** | **Changes the interface** to match what the client expects. |
| **Decorator** | Keeps the **same interface** but adds responsibilities/features. |
| **Proxy** | Keeps the **same interface** but controls access (security/caching). |

### 