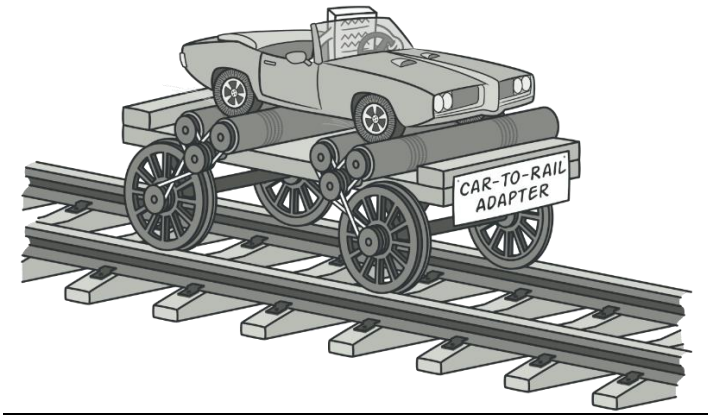
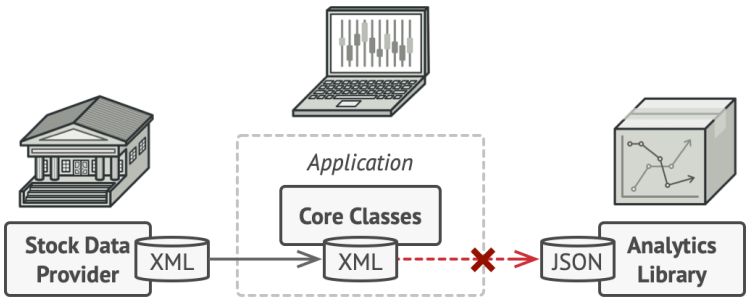
Adapter Design Pattern

Defⁿ:

- The *Adapter Design Pattern* is a *structural design pattern* that allows **incompatible interfaces** to work together. It acts as a *bridge* between two incompatible interfaces by *wrapping* an existing class with a **new interface**.
- In short, *Adapter* is a *structural design pattern* that allows objects with **incompatible interfaces** to *collaborate*.





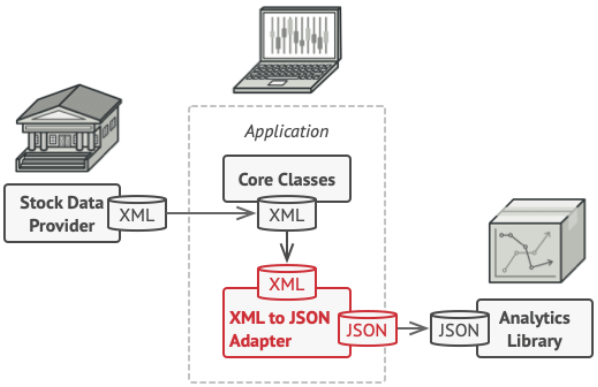
😞 **Problem:**

Imagine that you're creating a **stock market monitoring app**. The app downloads the **stock data** from multiple sources in *XML format* and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart **3rd-party analytics library**. But there's a catch: the **analytics library** only works with data in *JSON format*.

😊 **Solution:**

You can create *XML-to-JSON* **adapters** to communicate with the library. When an **adapter** receives a call, it *translates* the *incoming XML data* into a *JSON structure* and passes the call to the appropriate methods of a **wrapped analytics object**.

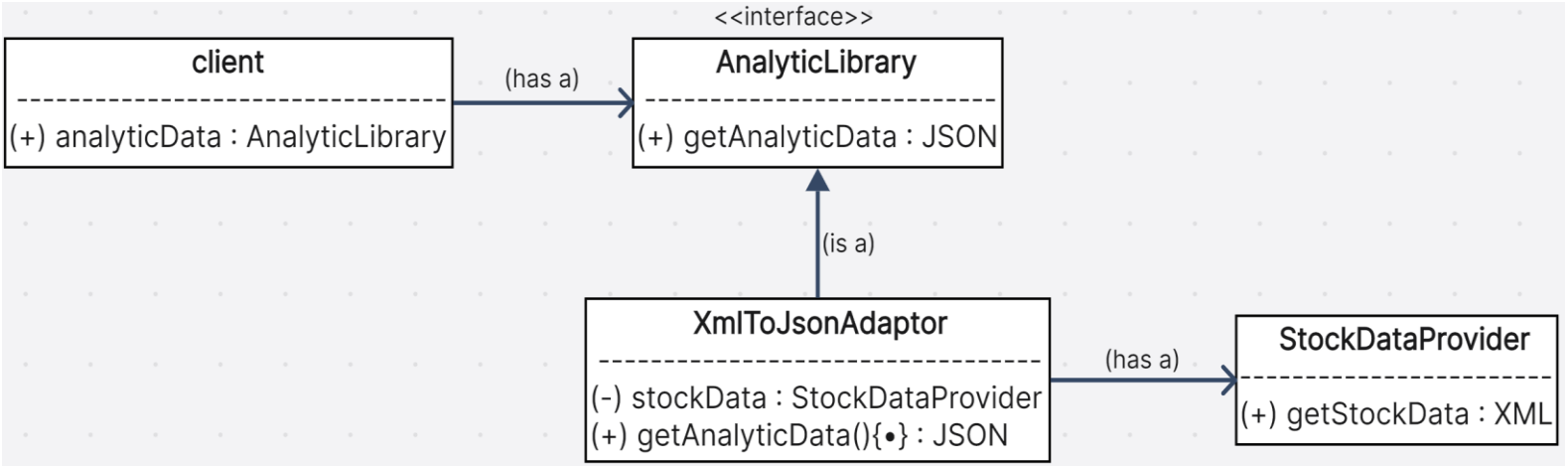


But I think simply we can make *a function* which takes an *XML* based **object** and returns *JSON* based **object**, then question arises **what's** the **need** of making *separate class* which implements JSON and takes XML as *composition*.

Answer is, the *Adapter Design Pattern* isn't just about *converting data*. It's about *adapting behaviour* and *interface* so that *client code* can work with *legacy* or *incompatible code* *without knowing about it*.

So basically, you'll build an *adapter* that converts *XML data* into *JSON format* --so the analytics library can understand it.

UML:



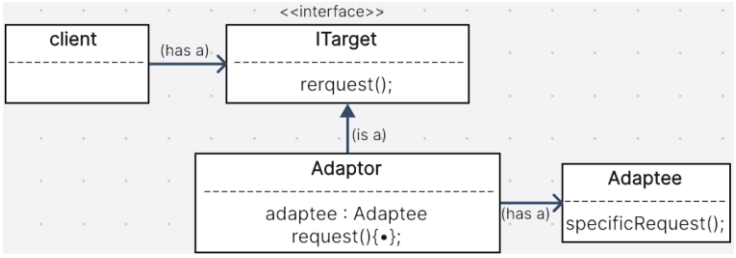
Code Link:

https://github.com/sibasundarj8/-System-Design-/tree/main/Codes/16___Adapter%20Design%20Pattern%20code

Types of Adapter design pattern:

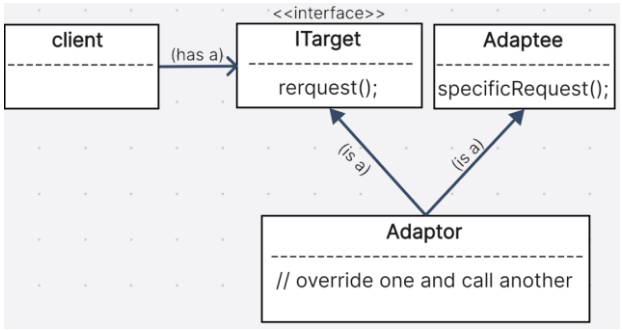
The **Adapter Design Pattern** can be applied in various ways depending on the **programming language** and the specific context.

1. Class Adapter (Inheritance-based)



- In this approach, the **adapter class** ***inherits*** from ***both*** the ***target interface*** (the one the client expects) and the ***adaptee*** (the existing class needing adaptation).
- Programming languages that allow ***multiple inheritance***, like **C++**, are more likely to use this technique.
- However, in languages like ***Java*** and ***C#***, which do not support ***multiple inheritance***, this approach is less frequently used.

2. Object Adapter (Composition-based)



- The ***object adapter*** employs ***composition*** instead of ***inheritance***. In this implementation, the adapter holds an ***instance*** of the ***adaptee*** and ***implements*** the ***target interface***.
- This approach is more **flexible** as it allows a ***single adapter*** to work with ***multiple adaptees*** and does not require the complexities of inheritance.
- The ***object adapter*** is widely used in languages like ***Java*** and ***C#***.

Use cases of Adapter Design Pattern:

1. Integrating ***Legacy Code*** with ***New Systems***
2. Using ***3rd-party*** or ***External Libraries***
3. ***Bridging Incompatible*** Interfaces
4. Hardware/Driver Abstractions