Problem

Imagine you have this cool app that keeps track of stock market data. The app currently fetches the stock info in a format called XML. Now, you want to add even cooler features using a fancy analytics tool, but it only understands data in JSON format.

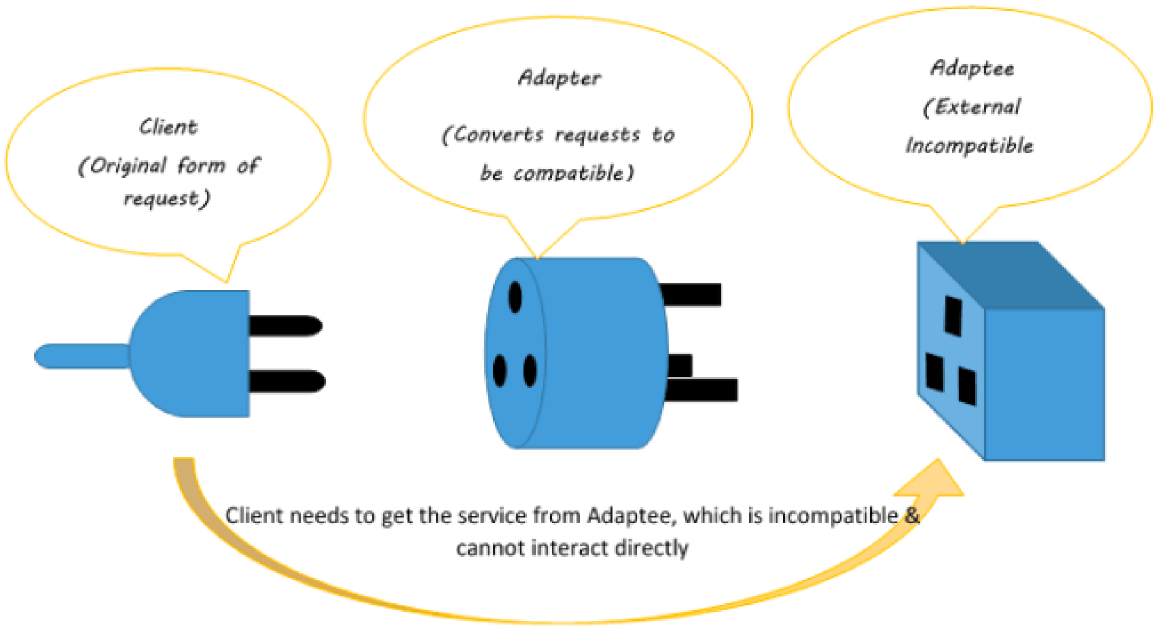
Now, you face a dilemma. You could tweak the analytics tool to understand XML, but that might mess up some existing parts of your app that rely on the tool. Plus, there's a chance you can't even tinker with the tool's inner workings since you don't have its source code.

Solution

Think of the Adapter design pattern like a language translator for objects. Imagine you have two classes that speak different "languages" (interfaces), and you want them to work together.

The Adapter comes into play as a friendly mediator. It understands both languages and helps them communicate. It's like a bridge that connects these two classes, making them understand each other's lingo without changing their core abilities.

So, the Adapter is the hero here, allowing two incompatible interfaces to work together smoothly, just like a translator helping people with different languages understand each other.


Client (Original form of request)
Adapter (Converts requests to be compatible)
Adaptee (External Incompatible
Client needs to get the service from Adaptee, which is incompatible & cannot interact directly

Components of Adapter Design Pattern

1. Target Interface

Description: Defines the interface expected by the client. It represents the set of operations that the client code can use.

Role: It’s the common interface that the client code interacts with.

2. Adaptee

Description: The existing class or system with an incompatible interface that needs to be integrated into the new system.

Role: It’s the class or system that the client code cannot directly use due to interface mismatches.

3. Adapter

Description: A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface.

Role: It acts as a bridge, adapting the interface of the adaptee to match the target interface.

4. Client

Description: The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter.

Role: It’s the code that benefits from the integration of the adaptee into the system through the adapter.

Let's break it down like we're talking about everyday stuff:

1. **Target Interface (Common Language):**

Description: It's like a menu at a restaurant. It lists what you can order, and you know what to expect.

Role: This is the common list (interface) that you (the customer) interact with.

2. Adaptee (Foreign Language Speaker):

Description: Imagine you're in a foreign country where people speak a different language. The Adaptee is like a local, speaking their own language that you don't understand.

Role: This is the class or system that talks a language you can't directly use.

3. Adapter (Translator):

Description: Now, you want to chat with the locals, but you need a translator. The Adapter is that translator who understands both your language (target interface) and the local language (adaptee).

Role: It acts as a bridge, helping the Adaptee communicate in a way that fits your expectations.

4. Client (You, the Traveler):

Description: You just want to enjoy your trip. You use the menu (target interface) to order, and the translator (adapter) makes sure your order gets through to the locals (adaptee).

Role: It's your code, blissfully unaware of the local language or translation process, just benefiting from the tasty dishes (results of the integration).