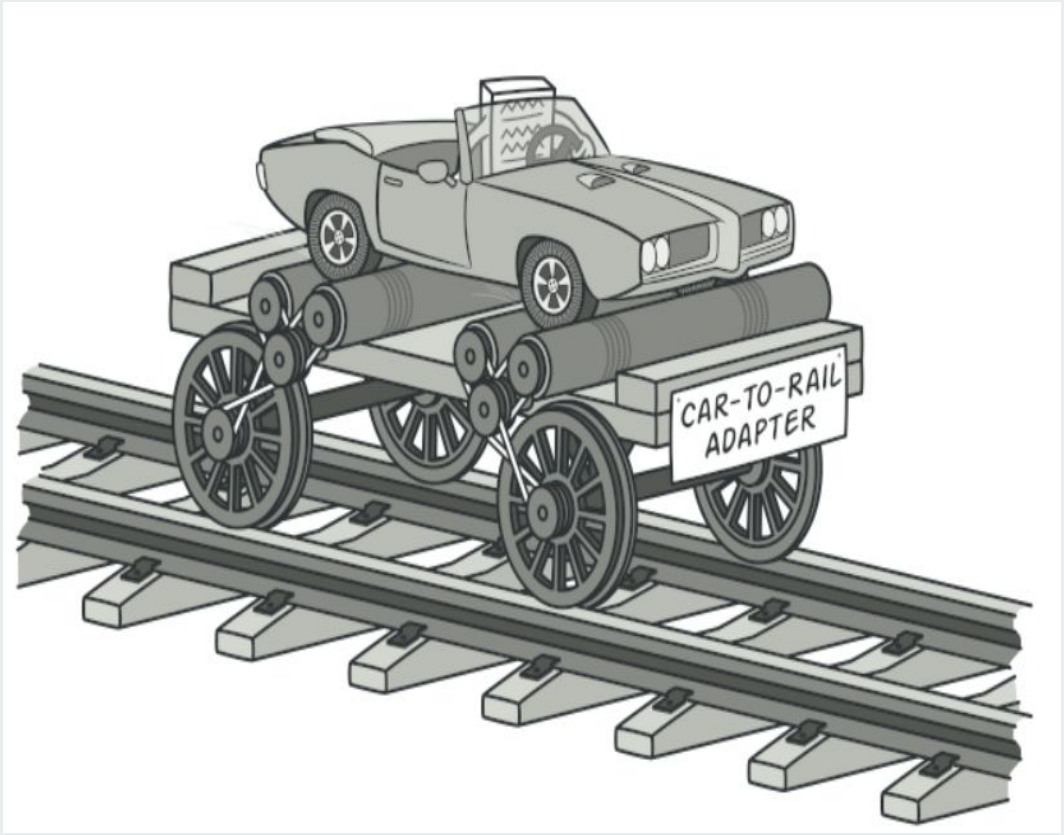
Adapter Design Pattern & Premature Optimization Antipattern

Group B



- **Structural design pattern**
- **Bridge between two incompatible interfaces**

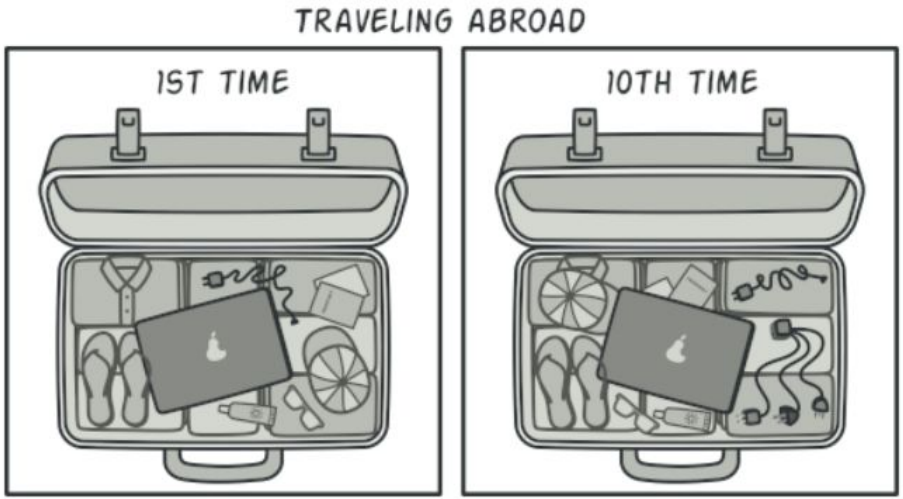
Adapter Design Pattern

- **Class which joins functionalities of independent or incompatible interfaces**
- **No need to rewrite the codes written before**
- Wraps an existing class with a new interface
- Convert an existing interface into another interface

Real life examples

- **Ex: Card reader (Adapter) between memory card and laptop.**
- **Ex: App that measure speed in US** in miles per hour (MPH), in UK in kilometers per hour (km/h). Adapter converts the values.

Real life examples



A suitcase before and after a trip abroad.

Problem

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

Solution

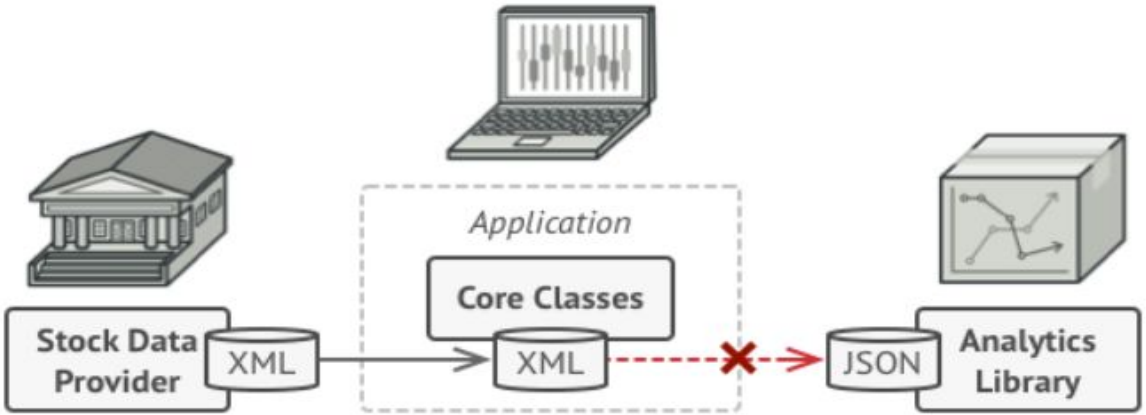
- Separate **adapter** class that converts the (incompatible) interface of a class (**adaptee**) into another interface (**target**)
- No change on already existing class
- Interfaces may be incompatible, but the inner functionality should suit the need

Solution

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

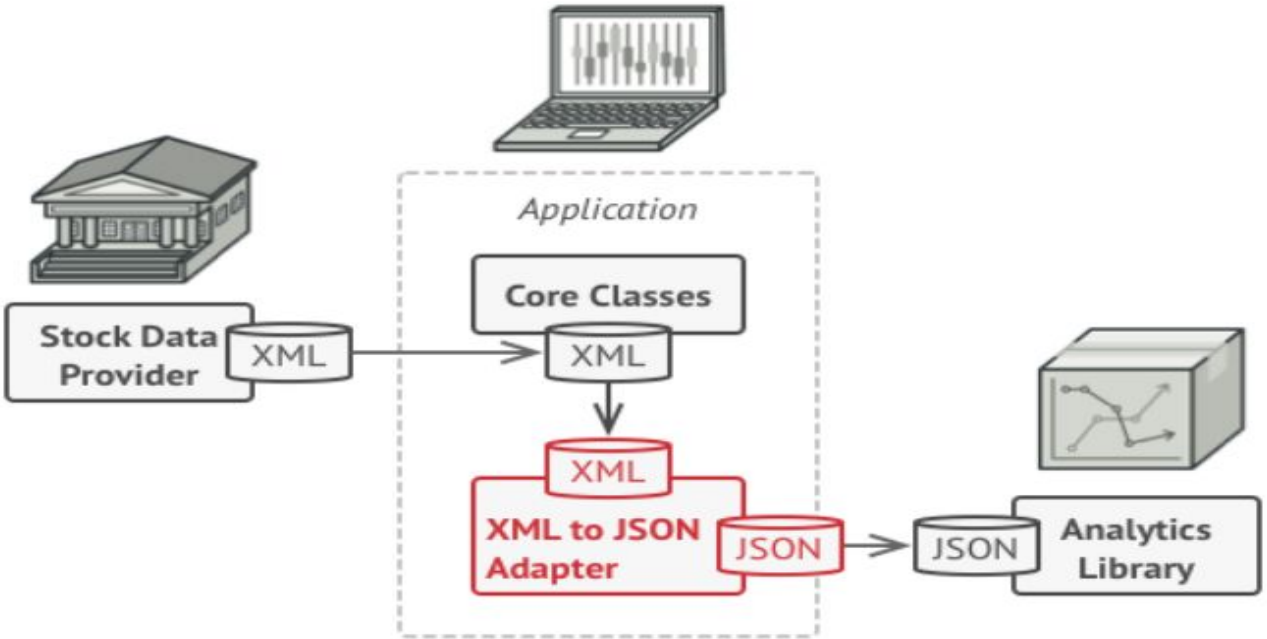
1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Problem



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

Solution

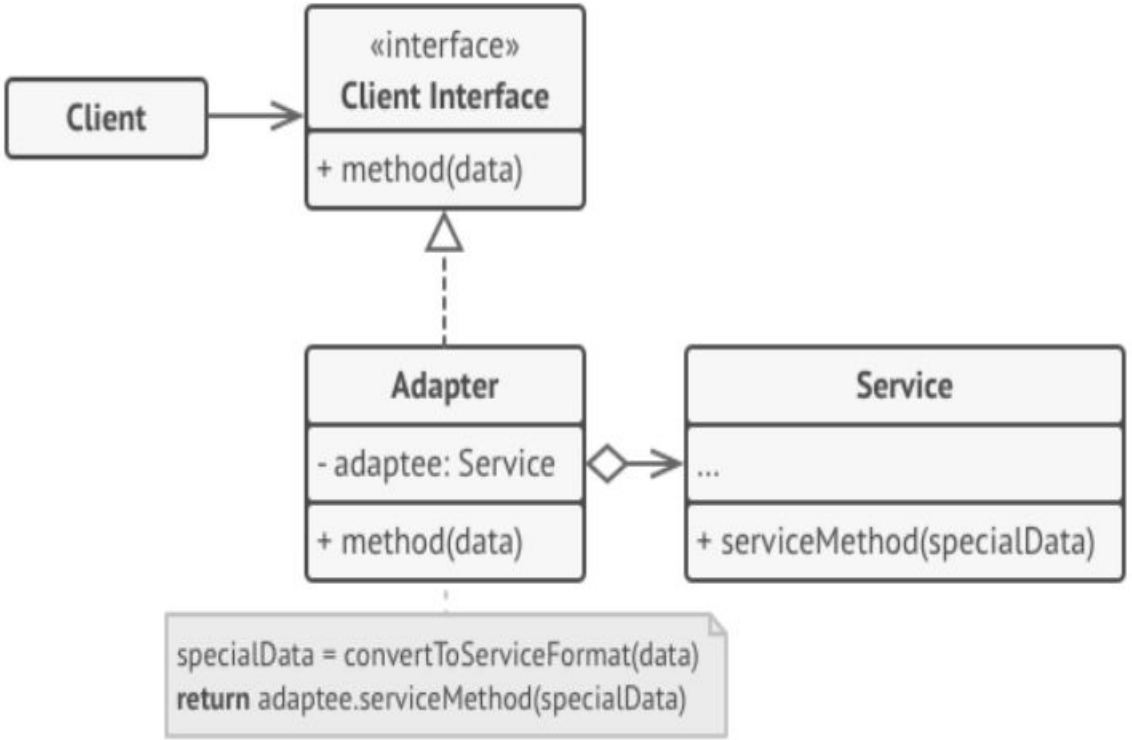

Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

How to Implement

1. Make sure that you have at least two classes with incompatible interfaces
2. Declare the client interface and describe how clients communicate with the service.
3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.
4. Add a field to the adapter class to store a reference to the service object.
5. One by one, implement all methods of the client interface in the adapter class.
6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

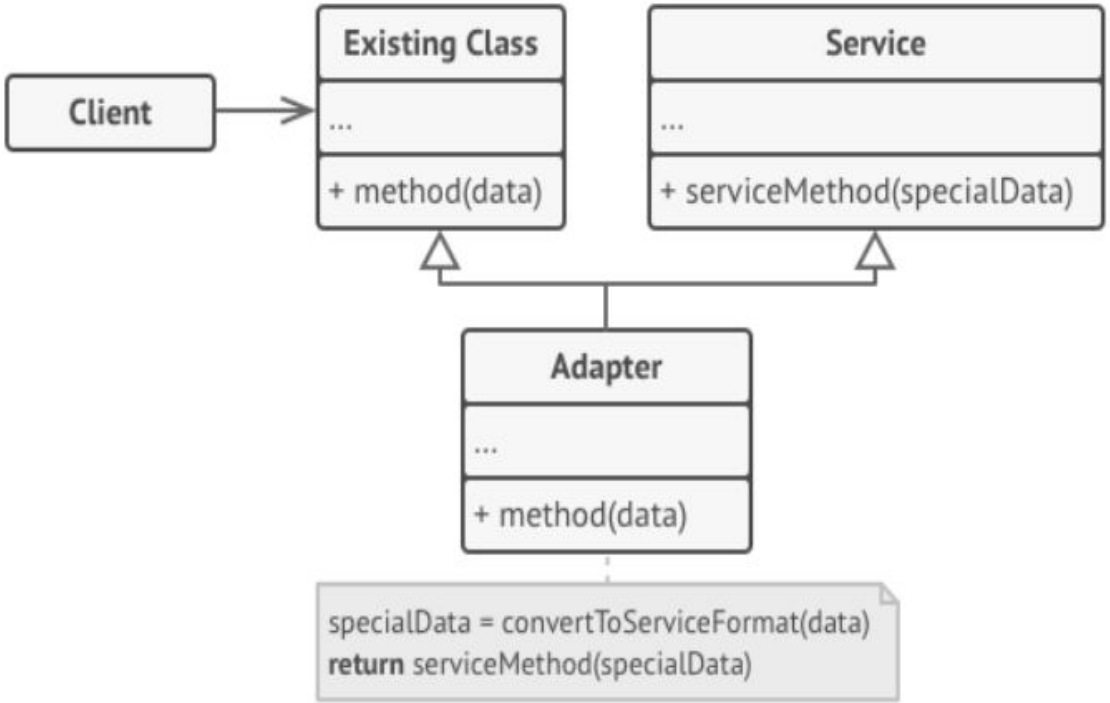
Object Adapter

- The adapter implements the interface of one object and wraps the other one.
- It can be implemented in all popular programming languages.



Class Adapter

- The adapter inherits interfaces from both objects at the same time.
- Only suitable for the programming languages that support multiple inheritance, such as C++



When to Use Adapter Pattern

- When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application. A suitable Adapter can be developed to make them compatible with each other
- When our application is not compatible with the interface that our client is expecting
- When we want to reuse legacy code in our application without making any modification in the original code

Pros

- *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
- The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.
- Prevents to duplicate code

Cons

- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes.
- Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Premature Optimisation

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

The Art of Computer Programming, Donald Knuth

Premature vs Basic Optimisation

These should not be mixed

Premature optimisation is done before more important parts of the software are done or possibly will be changed, that leads to waste of time for the developer.

Basic optimisation is done, while developing parts of the software that takes short time and with few changes that may lead to performance improvement of the code. Eg. adding a single comparison to end unnecessary code execution.

Educated conclusion

“Optimizing before you have enough information to make educated conclusions about where and how to do the optimization.”

Sahand Saba

Educated Conclusion

Writing well designed and readable software is preferred

Micro-optimise later

Give priority to known algorithms

Value of Time

“Observation #6: Software engineers have been led to believe that their time is more valuable than CPU time; therefore, wasting CPU cycles in order to reduce development time is always a win. They’ve forgotten, however, that the application users’ time is more valuable than their time.”

Randall Hyde

In real life

Eg.

Using a list, double-list, etc. in a code that would just hold maximum of few hundreds of items. You don't need to optimise it much. That's 97% Knuth is talking about.

But for some instances, where there are more data, or done a lot iterations over that data, even a small optimisation will be time efficient.