Adapter Design Pattern

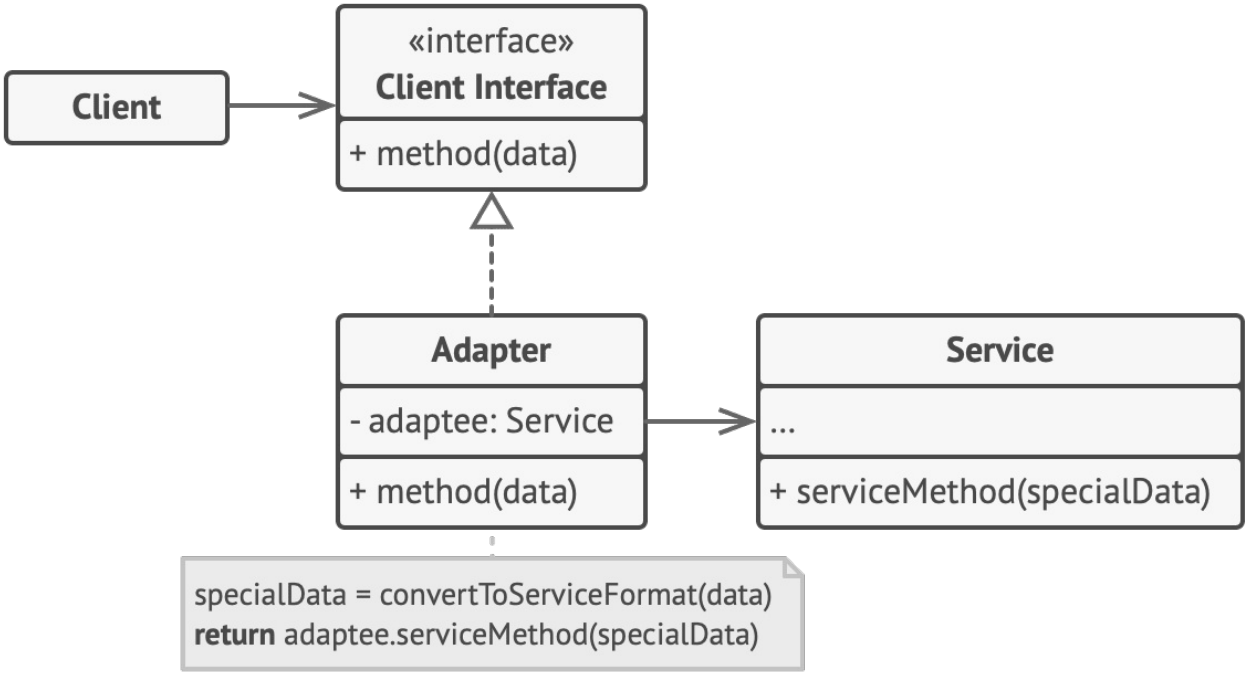
- Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Example - We have an existing object which provides the functionality that client needs. But the client code can't use this object because it expects an object with different interface.
- Problem: Suppose you are building a stock market monitoring app and you get all your data in xml format. Now you need to show good looking charts to end user and for that you decided to use a thord party library but that library accepts the data in JSON format.
- Solution: You can create an adapter which converts XML data to JSON data and then pass this to your third party library for charts.

Implementation considerations

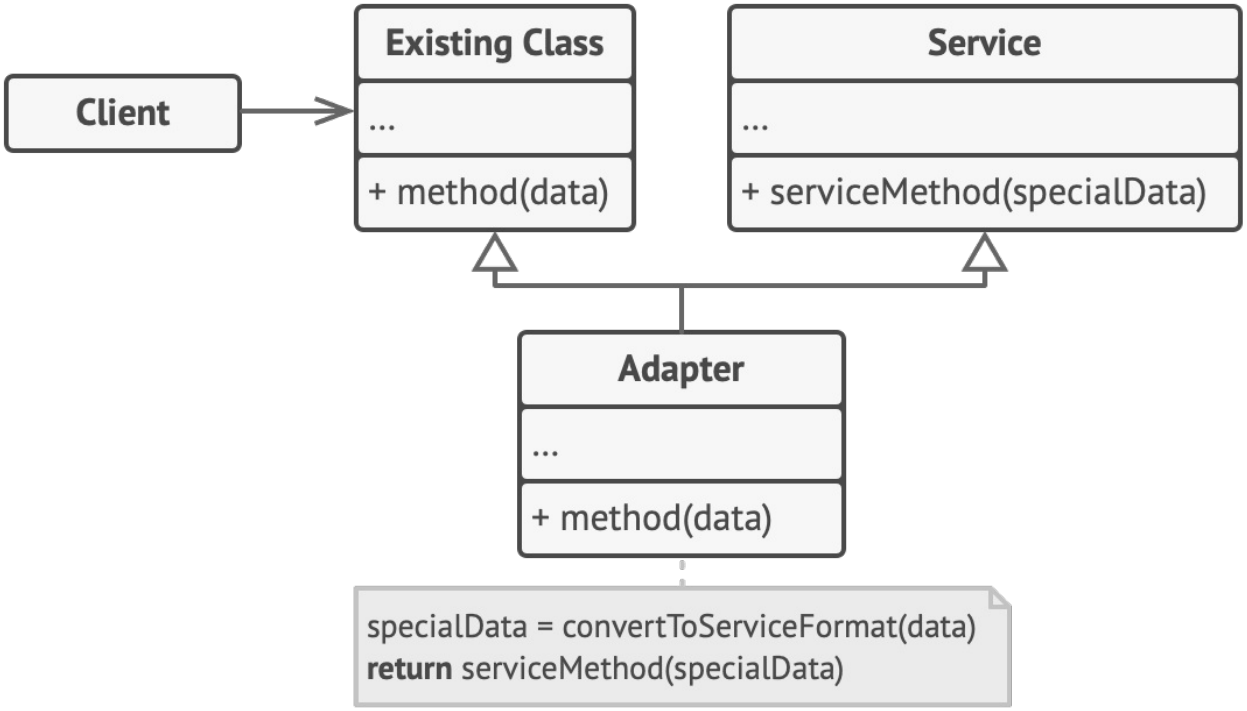
- Adapter must implement the interface expected by client.
- **Class Adapter (Two way adapter)** :
    - This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.
    - We are simply going to forward the method to another method inherited from adaptee.
    - This approach can only be implemented in programming languages that support multiple inheritance. (Java supports multiple inheritance using types i.e., interfaces)
    - Using class adapter allows you to override some of the adapter's behaviour. But we should avoid doing this as we can end up with adapter that behaves differently than adaptee. Fixing defects becomes hard in this case.
    - In java, this will not be possible if both target and adaptee are concrete classes.
- **Object Adapter** :
    - This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one.
    - Here, we are only going to implement target interface and accept adaptee as constructor argument in adapter.
    - It can be implemented in all popular programming languages.
    - This is more widely used as compared to Class Adapter.
    - Using object adapter allows you to potentially change the adaptee object to one of its subclasses.
- It can be tempting to do a lot of things in adapter besides simple interface translation. But this can result in an adapter showing different behaviour than the adapted class.

UML Diagrams

1. **Object Adapter**



2. **Class Adapter**



Applicability

- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

Pros and Cons

Pros	Cons
Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.	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.
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.	

References

- Composition: object A consists of objects B; A manages life cycle of B; B can’t live without A.
- https://refactoring.guru/design-patterns/adapter