Structural

Adapter/Wrapper

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

Solves

An *adapter* is a special object that converts the interface of one object so that another object can understand it.

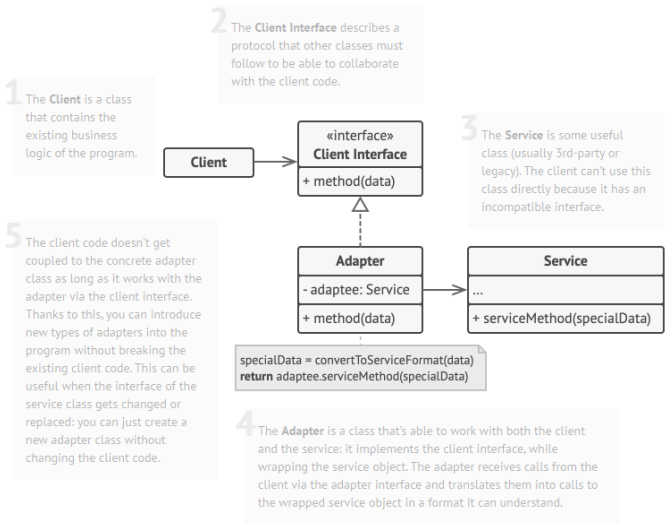
An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

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.

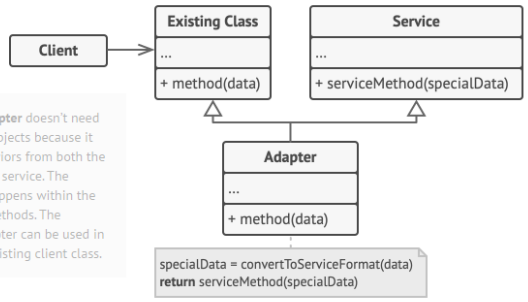
Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.

Structure



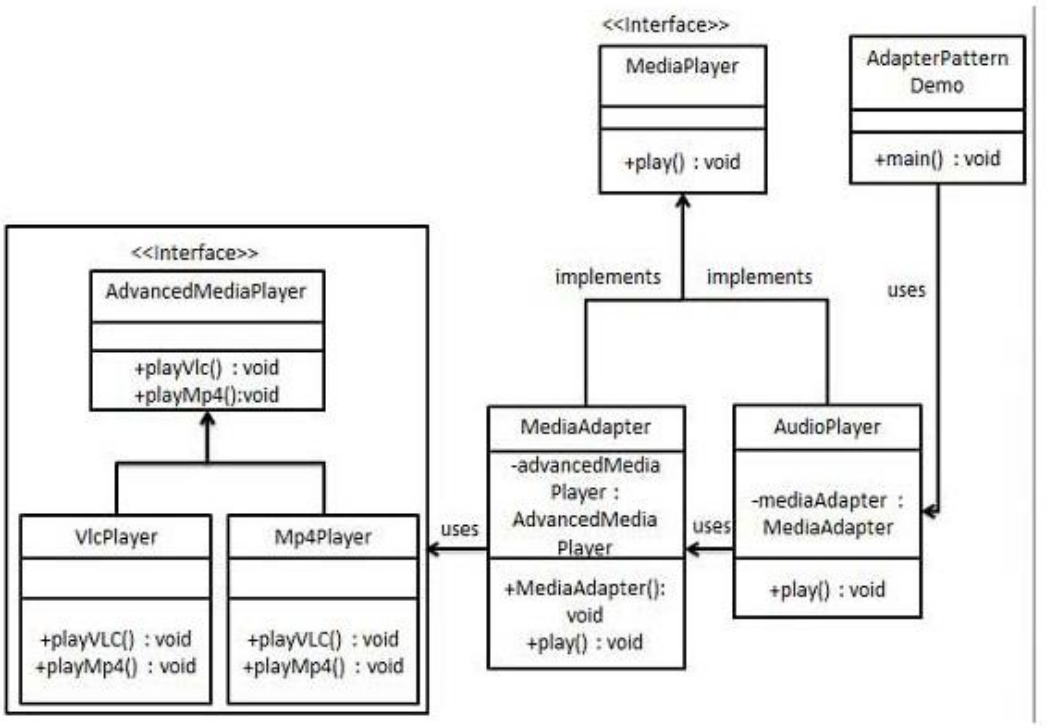
Class adapter

This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.



1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Code structure



Instances of Application

>Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.

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

>**Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.**

You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes

How to Implement

1. Make sure that you have at least two classes with incompatible interfaces:
    - A useful *service* class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
    - One or several *client* classes that would benefit from using the service class.
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. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.
5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.
6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Pros and Cons

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.

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.

Relationship with other patterns

- **Bridge** is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, **Adapter** is commonly used with an existing app to make some otherwise-incompatible classes work together nicely.
- **Adapter** changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*.
- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.

- **Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. *Adapter* usually wraps just one object, while *Facade* works with an entire subsystem of objects.
- **Bridge**, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.