29. Adapter

Adapter design pattern is used when you need to access some endpoint that is not compatible with the rest of your application, but you have no means of changing the endpoint. It has many real-life analogies.

Different regions of the world have different electric sockets. For example, a socket somewhere in continental Europe works with a plug that has two pins, while a British socket works with a triple-pin plug. So, if you bought some electric appliance in Britain and then traveled to continental Europe with it, you won't be able to plug it directly into a socket. You will need to get a socket adapter. This adapter will have two pins, which will allow it to be connected to a European socket, but it will also have a socket of its own, which will accept a British plug.

Adapter can be summarized as follows:

- There is a service that is routinely accessed by an application (we will call it **Service Implementation**).
- This service implements some interface (which we will call **Service Interface**).
- There is an endpoint that needs to be accessed by the same part of the application that isn't compatible with the **Service Interface**.
- To solve this problem, there is an **Adapter** class, which implements the **Service Interface**, but has some internal translation functionality to be able to access this incompatible endpoint.

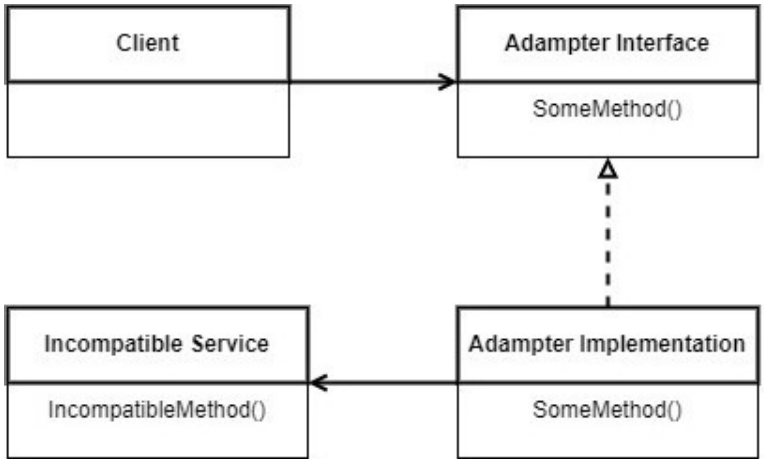

Figure 29.1 - Adapter UML diagram

For example, your application might be dealing with JSON data, but there's still an endpoint that deals with XML. And this endpoint cannot be changed, as many other services depend on it. So, if you want to be able to use this endpoint in the JSON-based part of your application, you need an **Adapter** that will translate between XML and JSON.

We will now go through an example implementation of the Adapter. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Structural_Patterns/Adapter

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Adapter implementation example

In this example, we will use the concept of electric sockets. We will mimic the use of a socket adapter in the code.

We will create a standard .NET console application. Then, we will add the `IElectricSocket.cs` file to it. The file will contain the following empty interface definition:

```
namespace Adapter_Demo;

internal interface IElectricSocket
{
}
```

Then, we will create an interface representing a socket plug. For this, we will add `ISocketPlug.cs` file with the following content:

```
namespace Adapter_Demo;

internal interface ISocketPlug
{
    void SelectSocket(IElectricSocket socket);
    void ConnectToSocket();
}
```

After this, we will need to add a specific interface that will allow us to connect to a European electric socket. This interface will inherit from `IElectricSocket` and will look like this:

```
namespace Adapter_Demo;


internal interface IEuropeanElectricSocket :
    IElectricSocket
{
    void ConnectTwoPins();
}
```

The main implementation of our interface will look like this:

```
namespace Adapter_Demo;


internal class EuropeanElectricSocket :
    IEuropeanElectricSocket
{
    public void ConnectTwoPins()
    {
        Console.WriteLine(
            "Double-pin plug has been successfully connec\
ted.");
    }
}
```

We will also have an interface and implementation for a British electric socket:

```
namespace Adapter_Demo;


internal interface IBritishElectricSocket :
    IElectricSocket
{
    void ConnectThreePins();
}


internal class BritishElectricSocket :
    IBritishElectricSocket
{
    public void ConnectThreePins()
    {
        Console.WriteLine(
            "Triple-pin plug has been successfully connec\
ted.");
    }
}
```

As you can see, even though it's still an electric socket, you can't connect a plug with two pins to it. You need three pins.

But the definition of our class that represents European socket plug will look as follows:

```
namespace Adapter_Demo;

internal class EuropeanSocketPlug :
    ISocketPlug
{
    private IEuropeanElectricSocket? europeanSocket;

    public void ConnectToSocket()
    {
        europeanSocket?.ConnectTwoPins();
    }

    public void SelectSocket(IElectricSocket socket)
    {
        if (socket is not IEuropeanElectricSocket)
        {
            throw new ArgumentException(
                "The European plug can only be connected \
to a European socket.");
        }

        europeanSocket = (IEuropeanElectricSocket)socket;
    }
}
```

As you can see, it can only work with classes that implement `IEuropeanElectricSocket`. `IBritishElectricSocket` implementations will not be compatible with it. And this is where we would need our adapter.

In our case, `IEuropeanElectricSocket` and `IBritishElectricSocket` are **Service Interfaces**, while `EuropeanElectricSocket` and `BritishElectricSocket` are **Service Implementations**. However, they represent different services that aren't compatible with one another. To solve this problem, we will need to add an **Adapter** object, which will represent another **Service Implementations** of

the `IEuropeanElectricSocket` interface, but internally will be able to work with `IBritishElectricSocket` implementations. And this is what our **Adapter** will look like:

```
namespace Adapter_Demo
{
    internal class SocketAdapter :
        IEuropeanElectricSocket, ISocketPlug
    {
        private IBritishElectricSocket? britishSocket;

        public void ConnectToSocket()
        {
            britishSocket?.ConnectThreePins();
        }

        public void ConnectTwoPins()
        {
            Console.WriteLine(
                "Double-pin plug has been successfully co\
nnected to the adapter.");
        }

        public void SelectSocket(
            IElectricSocket socket)
        {
            if (socket is not IBritishElectricSocket)
            {
                throw new ArgumentException(
                    "The adapter can only be connected to\
 a British socket.");
            }

            britishSocket = (IBritishElectricSocket) sock\
et;
        }
```

```
33      }
34  }
```

In our case, the **Adapter** acts both as `ISocketPlug` and `IEuropeanElectricSocket`, just like a real-life socket adapter would. It has the `ConnectTwoPins` method, which allows it to be used by an instance of the `EuropeanSocketPlug` class. But it also has `SelectSocket` and `ConnectToSocket` methods, which allow it to be connected to another socket. And this time, the socket it can connect to is an implementation of the `IBritishElectricSocket` interface.

Let's now see how this adapter works. To do so, we will replace the content of the `Program.cs` file with the following:

```
1   using Adapter_Demo;
2
3   var socketPlug = new EuropeanSocketPlug();
4   socketPlug.SelectSocket(
5       new EuropeanElectricSocket());
6   socketPlug.ConnectToSocket();
7
8   var adapter = new SocketAdapter();
9   adapter.SelectSocket(
10      new BritishElectricSocket());
11  adapter.ConnectToSocket();
12
13  socketPlug.SelectSocket(adapter);
14  socketPlug.ConnectToSocket();
15
16  Console.ReadKey();
```

So, we are first connecting a European socket plug into a European socket. Then we are connecting a socket adapter into a British socket. Finally, we are connecting the European socket plug into

the adapter. And this is what we get as the output if we run the application:

```
Double-pin plug has been successfully connected.
Triple-pin plug has been successfully connected.
Double-pin plug has been successfully connected to the adapter.
```

Figure 29.2 - Demonstration of adapter use

This concludes our overview of Adapter. Let's summarize its main benefits.

Benefits of using Adapter

Adapter design pattern has the following benefits:

- It allows you to make your application work with incompatible services without changing overall application logic.
- Any translation between an incompatible API and the rest of the application is confined to a single place, which makes the code easy to maintain.
- It works well when an incompatible API cannot be changed for any reason (it's maintained by a third party, many other services depend on it, etc.).

But, just like any other design pattern, Adapter has its caveats.

Caveats of using Adapter

Perhaps the main disadvantage of using Adapter is that it makes code more complicated. You need to add a suitable **Service Interface**, which may have to be more complex to make it work with an **Adapter** object. Also, the translation functionality itself may have to be very complex. Yes, the above example is fairly trivial. But in

a real-life situation, you may have to do some truly complex data transformation inside an **Adapter** object.

This is why, if you can, it's better to just change the incompatible service and make it compatible instead of using the Adapter design pattern. Use Adapter only if the service that you need to access is either a third-party service that cannot be changed or if too many other components depend on the existing API, which makes it too expensive to change.