10. Third-party components aren't directly compatible with your code

Imagine that you are working with a third-party library that returns data in a specific format. The problem is, however, that this is not the data format that the rest of your application works with.

For example, your code may only be working with JSON, while the third-party library only delivers data as XML. In this situation, the last thing that you would want to do is rewrite the rest of your code to make it compatible with XML, especially if you don't expect anything other than this library to use XML.

There is also another variety of this problem that you may encounter. Once again, you are dealing with a third-party library, but this time, its accessible interface is written differently from how the set of related functionality is written in your application. If this third-party library is meant to be able to replace your own components, ideally it should use the same signatures on its accessible members as your own classes are using. Once again, in this case, the last thing you want to do is include special cases into what was meant to be generic code.

It doesn't necessarily have to be a third-party library though. It could be your own legacy code that was written a while ago, when either a different philosophy was implemented or modern best practices weren't properly followed. In this case, you cannot just change the legacy components, as it has already been extensively tested and many other components within the system rely on it.

All these examples have one thing in common – you need to work with something that is structurally different from the rest of your code, but making changes to your own code is not desirable.

Suitable design patterns

Adapter

You can think of an Adapter as a wrapper class for whatever component you would want to use in your code that is currently incompatible with it. The Adapter class will be accessible by using exactly the same interfaces that are normally used in your code, while internally it will be calling methods on the external component. If it needs to transfer the data, it will do so in a format that is compatible with the rest of your code base.

Basically, the Adapter class is analogous to a real-life physical adapter, such as an electric socket plug adapter. As you may know, the US, UK, and continental Europe use different electric sockets. So if you are traveling from the US to Europe, you won't be able to just plug your electronic devices in. You will have to get an adapter that has the same input configuration as a wall socket in the US but has the same output pin configuration as a European socket plug.

Why would you use Adapter

- Adapter allows you to isolate interface conversion functionality to only a single place in your code.
- Open-closed principle is enforced
- The access point into the immutable external component is standardized along with the rest of your code.