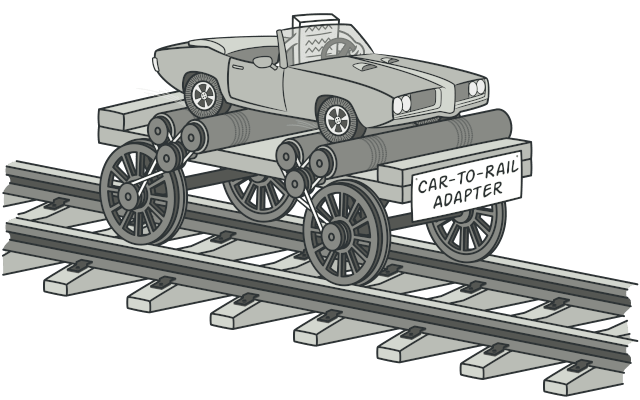
**Adapter**

(It’s also called: *Wrapper*)

1)Introduction:

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



**2)Define:**

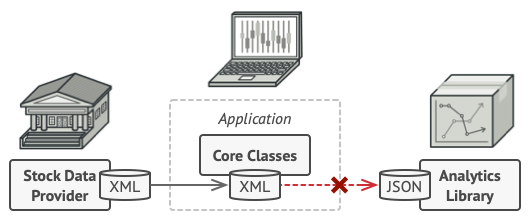
Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

3)Purpose of use:

a)Problem:

Imagine that you’re creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there’s a catch: the analytics library only works with data in JSON format.



b)Solution

Use adpater

Why?

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.

4)Architecture:

According adapter ‘s implement, there are two way to perform Adapter Pattern.

a)Object adapter-Composition:

This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one.

Diagram

Description automatically generated

Components:

1. Client: is a class that contains the existing business logic of the program.
2. Client Interface: describes a protocol that other classes must follow to be able to collaborate with the client code.
3. Service: is some useful class (usually 3rd-party or legacy). The client can’t use this class directly because it has an incompatible interface
4. Adapter is a class that’s able to work with both the client and the service.The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

\*Note:

* The client code doesn’t get coupled to the concrete adapter class as long as it works with the adapter via the client interface.
* Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code.
* This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

**b. Class Adapter – Inheritance**

This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.

Diagram

Description automatically generated

Component:

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.

Difference between Class Adapter and Object Adapter

The main difference is that with class adapter we subclass the Target and the Adaptee, while the object adapter uses composition to pass requests to an adaptee. Object Adapters and Class Adapters use two different means of adapting the adaptee: composition versus inheritance.

**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

* *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.

## Relations 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.