Adapter 22

You are going to London and want to plug your hairdryer into the socket – since the German plug does not fit into the British socket, you need an adapter. I present you the pattern on the level of single classes. In fact, it can be scaled – the principle is the same if you adapt whole subsystems.

22.1 An Introductory Example

Let's generalize the plug example. You have two systems – the hairdryer and the plug socket – which have to work together. The "must" means that you have no other option in London than to dock with the UK socket. In IT, "must" can mean that you want to use a system or class that represents something unique or highly complex, such as a premium or deadline calculation in the insurance industry. A complex adapter is also JDBC, for example. The interaction of the systems that "must" work together is only disturbed by the fact that the client expects a different interface than the system to be used declares. The adapter has the task of mediating between the two systems. The GoF describes the adapter in two ways: object-based and class-based. Both designs are discussed in this chapter.

22.2 A Class-Based Design

What is the task to be solved in this chapter? Take a look at the Adapter_ClassBased sample project. You have purchased a third-party library that sorts numbers using a blazing fast algorithm.

```
    public class SorterExternalProduct {
        List<Integer> sort(List<Integer> numberList) {
            // … black box
        }
    }
```

All you know about this library is that you need to pass an integer list as a parameter and get back a sorted integer list. However, your software is programmed to expect a sorter object with the following interface.

```
    public interface Sorter {
        int[] sort(int... numbers);
    }
```

The source code of the third-party library is not available to you, so you cannot make any changes to it. Your own application is an established tested procedure that has already been delivered to many customers – it must not be changed under any circumstances. So you have to design an adapter that mediates between the foreign library and your application. What does such an adapter look like? The class-based approach is to let the adapter inherit from the foreign library. At the same time, the adapter takes on the role of a sorter object, that is, it implements the expected interface.

```
    public class SorterAdapter
            extends SorterFremdprodukt implements Sorter {

        @Override
        public int[] sort(int[] numbers) {
            // … we'll deal with later
        }
    }
```

The client can now send its message as it expects.

```
    public class Client {


        public static void main(String[] args) {
            final int[] numbers = { 9, 4, 7, 3, 5 };
            var sorter = new SorterAdapter();
            final var sorted = sorter.sort(numbers);

            for (var number : sorted)
                System.out.println(number);
        }
    }
```

The adapter's sort method converts the array of numbers into a list and calls the correct method on the foreign library. So the adapter pretends to be a sorter, but is actually a subclass of the foreign library. Now here is the entire code of the adapter:

```
public class SorterAdapter
        extends SorterFremdprodukt implements Sorter {

    @Override
    public int[] sort(int[] numbers) {
        int z = new int[numbers.length];
        List<Integer> numberList = new ArrayList<>();
        for (var number : numbers)
            numberList.add(number);
        List<Integer> sortedList = sort(numberList);
        for (var i = 0; i < sortedList.size(); i++)
            z[i] = sortedList.get(i);
        return z;
    }
}
```

The GoF describes the class-based approach using multiple inheritance (in C++). Since Java does not know multiple inheritance, you have to make the compromise that you declare the expected interface by an interface.

22.3 An Object-Based Design

If the class-based design has already solved the problem, we could actually close the chapter, right? There are two reasons against being satisfied too quickly. The first reason is that one design principle is to prefer composition to inheritance. The second reason is quite trivial – the provider of the third-party library has marked the class as final in an update.

```
public final class SorterExternalProduct {
    // … abridged
}
```

The adapter of the previous solution can now no longer be used. Your only option is to program an object-based adapter. How do you do this? You create an attribute that holds a reference to an object in the foreign library. The sort method calls the method `sort()` on this attribute. You can find this variant in the Adapter_ObjectBased sample project.

```
public class SorterAdapter implements Sorter {
    private final SorterExternalProduct externalProduct =
                                    new SorterExternalProduct();
```

```
        @Override
        public int[] sort(int[] numbers) {
            List<Integer> numberList = new ArrayList<>();
            for (var number : numbers)
                numberList.add(number);
            var sortedList =
                                externalProduct.sort(numberList);
            for (var i = 0; i < sortedList.size(); i++)
                numbers[i] = sortedList.get(i);
            return pay;
        }
    }
```

You are now familiar with both approaches – object-based and class-based. In the following section, you will look at the advantages and disadvantages of both approaches.

22.4 Criticism of the Adapter Pattern

Please look again at the clients of the two approaches. You can see that the client knows the target interface and the adapter. Everything beyond the adapter is not visible to the client. Therefore, there may be another class or an entire system behind the client. Since the client does not know the system behind the adapter, this can be replaced without any problems. The only important thing is that the adapter is implemented correctly. In Sect. 21.4, you looked at the Law of Demeter. The adapter pattern also helps you to develop systems where classes communicate only with close friends.

In this context, let me distinguish the adapter from the facade. In both cases, you have a system that a client wants to access. It does not access this system directly, but via another abstraction. The main difference is the target direction. In the case of the facade, the goal was to facilitate access to a complex or complicated system. The adapter has the task of enabling two systems to communicate with each other in the first place. This brings up another obvious difference: who creates the abstraction? In the case of the facade, it was clear that the provider of the system must create a facade; it provides simplified handling. With the adapter, things are usually different. The vendor of a system invests a lot of time and effort in creating his software. He develops an interface to which the client can direct its requests. With that, his job is done. If the client needs another interface, he has to create it and define an adapter.

An adapter is usually a class that has been written and optimized for a single use case. The adapter I developed in the example above is difficult to reuse, which is sometimes described as a disadvantage in the literature. You will undoubtedly have to consider whether it is better to refactor one of the interfaces in such cases. However, you cannot rework the interface if the adapter allows you to access the Internet or a database (JDBC).

If you choose the class-based approach, the adapter can easily override methods of the classes to be adapted. However, it is then bound to a class because it has itself become a subclass of the class to be adapted. A new adapter must be developed for each subclass of the classes to be adapted. What about the object-based approach? In Sect. 2.4, I touched on Liskov's substitution principle, which states that a subclass should behave exactly like its base class; in other words, a subclass must be able to represent its base class. If an inheritance hierarchy takes this principle into account, it is possible for an object-based adapter to be used not only for a class but also for its subclasses.

Background Information

Did you know that the adapter pattern can already be traced back to the Brothers Grimm? Surely you know the fairy tale Little Red Riding Hood. The wolf disguises himself as grandmother; Little Red Riding Hood sees an object lying in bed that corresponds to the interface grandmother, and at first does not suspect anything. Later, however, it realizes that a class-based WolfAdapter is hidden behind the interface – otherwise it would never have come to the legendary quote: "Grandmother, why do you have such a horribly big mouth?" So the adapter is obviously poorly implemented and throws a WolfException.

The fairy tale can teach us that it's always a bad idea for people to concoct their own solutions without checking with IT first.

22.5 A Labeling Fraud

You will find a large number of classes in the class library that have "adapter" in their name. In Swing, for example, there are `MouseAdapter`, `FocusAdapter`, etc. The `MouseAdapter` class implements the `MouseListener`, `MouseMotionListener`, and `MouseWheelListener` interfaces. All methods are overridden empty, so you can define a `MouseListener` without having to override all methods yourself. If you want to override a method from the `MouseMotionListener` interface, expand the MouseAdapter and specifically override only that one method.

Is this an adapter in the sense of the Adapter Pattern? No – the `MouseAdapter` class does not bring you any added value, nor does it mediate between two systems. It does, however, make it easier for you to deal with the interfaces mentioned. Simplified access to a subsystem was the hallmark of the facade. So the `MouseAdapter` class should be called `MouseFacade`. So not everywhere that says adapter on it is adapter in it.

22.6 Adapter – The UML Diagram

Figures 22.1 and 22.2 show a comparison of the UML diagrams from the two sample projects Adapter_Class-Based and Adapter_Object-Based.

22.7 Summary

Go through the chapter again in key words:

- There are two systems whose interfaces are incompatible.
- An adapter mediates between the two systems.
- It converts the interface of the system to be adapted into the one expected by the client.
- The original classes are not changed.
- An adapter is written for exactly one use case.
- Client and adapted system are loosely coupled.
- Both systems can be interchanged independently as long as the adapter fits.
- The adapter exists in two forms: Object-based and class-based.

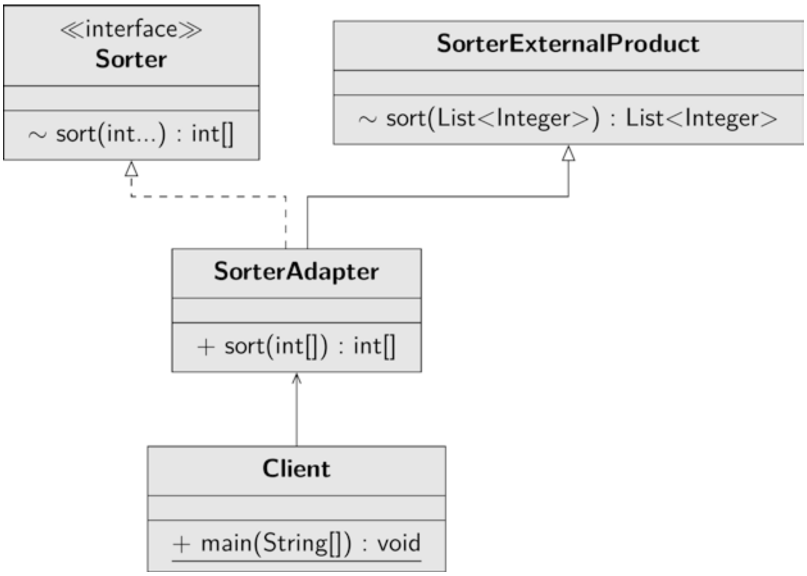

Fig. 22.1 UML diagram of the adapter pattern (example project Adapter_ClassBased)

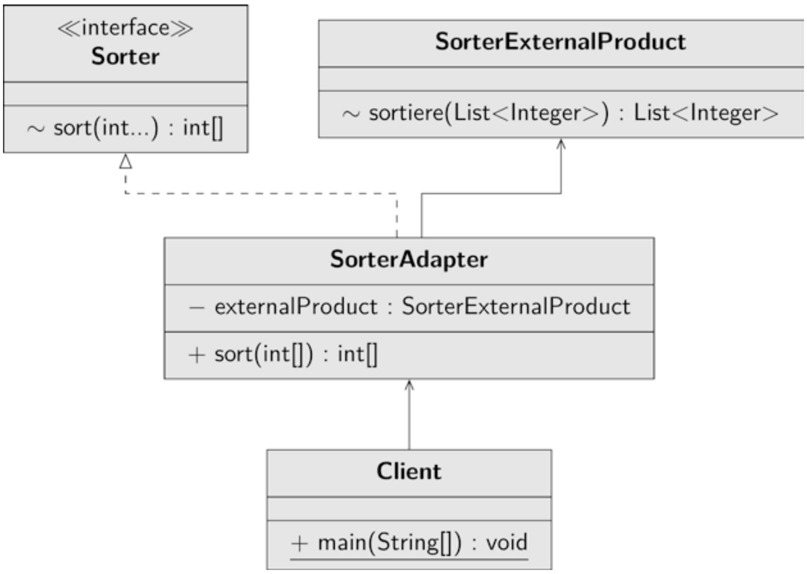

Fig. 22.2 UML diagram of the adapter pattern (example project Adapter_Object-based)

22.8 Description of Purpose

The Gang of Four describes the purpose of the "Adapter" pattern as follows:

> Adapt the interface of a class to another interface expected by its clients. The adapter pattern lets classes work together that would otherwise be unable to do so because of incompatible interfaces.