Adapter Pattern

<https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/>

We use adapters in our daily lives. The moment you plugin your mobile handset or your laptop to a socket for charging, an adapter is at work. What the adapter does is to make the socket that produces 120 V (or 220 V for European standard) and the mobile device that requires 4 V work together. Similarly, by using the adapter pattern in the programming world, you can make incompatible interfaces work together.

## Participants of the Adapter Pattern

To understand how the adapter pattern works, consider an existing text formatting application comprising of a TextFormattable interface and a NewLineFormatter implementation class. A client provides a string to format with a call to the formatText(String text) method declared in the interface. The implementation class returns a new string by replacing the periods with new line characters.

As an enhancement, the application now needs to support CSV formatting so that the text can be read and edited in a spreadsheet software. The new feature has been implemented and delivered by an external vendor as a CsvFormattable interface along with a CsvFormatter implementation class.

To integrate the new feature in the existing application, several issues require addressing. Primarily, the interface against which the new requirements are implemented has a different structure with different methods, and this is not what the clients of the application expect.

One approach to address such incompatibilities is to ask the vendor to supply a new version with a compatible interface. Another is to update the structure of the existing interface to accommodate the new requirement. Both the approaches will not only result in major rework but also carry the risk of violating the[SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) programming principles, specifically [Single Responsibility Principle](https://springframework.guru/single-responsibility-principle/) and [Interface Segregation Principle](https://springframework.guru/interface-segregation-principle/). In such situations, the adapter pattern comes to the rescue.

Using the adapter pattern, you can create a class, say CsvAdapterImpl that will act as an adapter to make both the incompatible interfaces (TextFormattable and CsvFormattable) work together.

There are two variants of adapters: Object adapter that relies on composition and class adapter that relies on inheritance. As Java does not support multiple inheritance, you have to use object adapter when there are multiple classes that the adapter needs to address. Also, the approach to [favors composition over inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance) should be the driving factor for using object adapters in Java.

We can summarize the participants of the adapter pattern in the context of the text formatting example, as:

* Target (TextFormattable): The existing interface that clients communicate with.
* Adaptee (CsvFormattable): The new incompatible interface that needs adapting.
* Adapter (CsvAdapterImpl): A class that adapts the Adaptee to the Target.
* Client: Communicates with the Target.

## Applying the Adapter Pattern

To apply the adapter pattern to the text formatting example, let’s look at the existing Target interface and its implementation class.

**TextFormattable.java**

public interface TextFormattable {

String formatText(String text);

}

**NewLineFormatter.java**

public class NewLineFormatter implements TextFormattable{

    @Override

    public String formatText(String text)

    {

        String formattedText=text.replace(".","\n");

        return formattedText;

    }

}

In the example above, the TextFormattable interface declares a single formatText() method that the NewLineFormatter class overrides to return a string formatted with new line characters.

Next, let’s look at the Adaptee interface and its implementation class.

**CsvFormattable.java**

package guru.springframework.gof.adapter.adaptee;

public interface CsvFormattable {

String formatCsvText(String text);

}

**CsvFormatter.java**

public class CsvFormatter implements CsvFormattable{

@Override

public String formatCsvText(String text){

String formattedText=text.replace(".",",");

return formattedText;

}

}

The CsvFormattable interface declares a formatCsvText() method that the CsvFormatterclass overrides to return a string formatted as comma separated values. Coming to the main part, we will now write the Adapter class.

**CsvAdapterImpl.java**

package guru.springframework.gof.adapter.csvadapter;

import guru.springframework.gof.adapter.adaptee.CsvFormattable;

import guru.springframework.gof.adapter.source.TextFormattable;

public class CsvAdapterImpl implements TextFormattable {

CsvFormattable csvFormatter;

public CsvAdapterImpl(CsvFormattable csvFormatter){

this.csvFormatter=csvFormatter;

}

@Override

public String formatText(String text)

{

String formattedText=csvFormatter.formatCsvText(text);

return formattedText;

}

}

In the CsvAdapterImpl class above, we implemented the TextFormattable interface, which is the Target. We then declared the Adaptee type (CsvFormattable) as a field and initialized it in the constructor. In the overridden formatText() method, we made a call to the formatCsvText()method, and returned back the CSV formatted string to the caller. Let’s now write a unit test for the example.

**NewLineFormatterTest.java**

import guru.springframework.gof.adapter.adaptee.CsvFormattable;

import guru.springframework.gof.adapter.adaptee.CsvFormatter;

import guru.springframework.gof.adapter.csvadapter.CsvAdapterImpl;

import org.junit.Test;

import static org.junit.Assert.\*;

public class NewLineFormatterTest {

@Test

public void testFormatText() throws Exception {

String testString=" Formatting line 1. Formatting line 2. Formatting line 3.";

TextFormattable newLineFormatter=new NewLineFormatter();

String resultString = newLineFormatter.formatText(testString);

System.out.println(resultString);

CsvFormattable csvFormatter=new CsvFormatter();

TextFormattable csvAdapter=new CsvAdapterImpl(csvFormatter);

String resultCsvString=csvAdapter.formatText(testString);

System.out.println(resultCsvString);

}

}

In Line 17 of the test code above, we called the formatText() method of TextFormattable to format text without using the adapter. It is in Line 20 – Line 22 where we used the adapter. We created a CsvAdapterImpl object passing a CsvFormatter object to its constructor. We then called the formatText() method, which at runtime got forwarded to a call to formatCsvTex() on CsvFormatter.

# Conclusion

Some programmers argue that the Adapter pattern is a workaround for a system, which was not well designed keeping into considerations new possibilities. While this is true to some extent, we cannot expect an enterprise application, which will often have a pluggable architecture, to be designed considering all components that might get added in future.  
In enterprise application development, it is likely that you might need to hook in other libraries, APIs, and “off the shelf” components, and if they are not aligned with the existing system, put the adapter pattern to use. After all, being a core GOF pattern, it is a tested and proven solution used over a long period of time.