# Adapter Pattern

Preslav Gerchev 2487136  
Dimitar Vikentiev 2464020

# Introduction

In week 5 we had to develop a Analog and Digital transmitting system using the Adapter Pattern.  
The following document will reflect on the project and on the pattern itself- positive and negative consequences, reusability, maintainability, extensibility.

# The Pattern

The formal definition for the pattern is as following**:**

**“The adapter pattern is a**[**software design pattern**](https://en.wikipedia.org/wiki/Software_design_pattern)**that allows the**[**interface**](https://en.wikipedia.org/wiki/Interface_(computer_science))**of an existing**[**class**](https://en.wikipedia.org/wiki/Class_(computer_science))**to be used from another interface. It is often used to make existing classes work with others without modifying their**[**source code**](https://en.wikipedia.org/wiki/Source_code)**.”**

# Reusability

The adapter pattern strongly promotes reusability. It allows for using an already created interface and “transform it” into another interface via an adapter. The pattern is mostly used to make two otherwise incompatible interfaces work together. However their functionality should be somewhat similar. An example would be the digital and the analog transmitter from our application – they both have the same functionality (transmitting data) but they do it differently. The adapter helps us wrap the analog interface (the adaptee) and treat it as a digital one (the target).

The pattern’s strongest advantage is that it connects *already created* interfaces – it removes the need for changing them or creating new ones.

# Extensibility

The Adapter pattern also improves the code in terms of extensibility. It extends an already created interface by using an adapter so that same interface can be treated differently.

One of the pattern’s disadvantages is that for each “target-adaptee” pair we’d need a new adapter implementation since the adapter holds a concrete reference to the adaptee interface in order to be able to delegate the responsibility of the target’s operations to the adaptee (something very similar to the Strategy pattern – this will be discussed later).

Another disadvantage is that since we’re using composition and not inheritance (staying true to the design pattern principles) if there’s a change in the adaptee’s interface that would also have to be reflected in the adapter. The code will have to be modified in order to expose the new operations.

An advantage of the Adapter pattern is that since we’re coding to a supertype (or in other words – interface and also staying true to another design principle) we are also “adapting” all the adaptee’s subclasses and they can be used with the adapter.

# Maintainability

The adapter is relatively easy to maintain. Since we’re using composition and delegating the responsibility to the adaptee the adapter doesn’t care if something changes in the adaptee’s implementation (as long as there are no new methods in the interface’s contract of course).

For example if the logic for one of the operations is changed the adapter will not even notice – its only job is to bring together adaptee to the target and use the adaptee as an object of the target’s interface type.

# Extra

## Relation with the Strategy Pattern

The Adapter pattern is very similar to the Strategy pattern that we implemented in week 1 since it relies on composition and not inheritance. In most parts it is the same except for the actual goal of the patterns – the adapter pattern’s goal is to bring together 2 otherwise non-related interfaces and the strategy pattern aims to change the behavior of an interface during runtime choosing from a family of algorithms.

## What is the Adapter’s responsibility?

There are two scenarios that could happen with the adapter and its responsibilities. One of those is to delegate the target’s request to the adaptee. In other words to simply call the adaptee’s operations in those of the target. This is the scenario when the interfaces we’re adapting are similar and there is no need to process the data from the adaptee. An example is given below of the adapter’s implementation.

*private Adaptee adaptee;*

*public void TargetOperation()*

*{  
 adaptee.TargetOperation();*

}

The other scenario is where the adapter has a slightly bigger responsibility – to process the data from the adaptee in some way so that it can be comprehensible by the target. Our application shows exactly that: since an analog data cannot be read by digital systems it has to be converted in some way to digital way – and that is the adapter’s responsibility. An example is given below:

*private Adaptee adaptee;*

*public void TargetOperation()*

*{*

*ProcessAdapteeDataAndMakeItReadableForTarget();*

}

# Summary

The adapter pattern finds its usage in applications where we already have two existing non-related interfaces and we want to bring them together and make them compatible. Its main advantage is that it adapts *already existing and implemented* interfaces. It strongly promotes reusability and is relatively easy to implement and maintain.