**Structural**

**Adapter**

The Adapter design pattern is a structural pattern that allows incompatible interfaces to work together. By doing so, we allow objects from different interfaces to exchange data.

We should use the Adapter class whenever we want to work with the existing class but its interface is not compatible with the rest of our code. Basically, the Adapter pattern is a middle-layer which serves as a translator between the code implemented in our project and some third party class or any other class with a different interface.

Furthermore, we should use the Adapter when we want to reuse existing classes from our project but they lack a common functionality. By using the Adapter pattern in this case, we don’t need to extend each class separately and create a redundant code.

## *Conclusion*

The Adapter pattern is pretty common in the C# world and it is quite used when we have to adapt some existing classes to a new interface. It can increase a code complexity by adding additional classes (adapters) but it is worth an effort for sure.

**Decorator**

A Decorator is a structural design pattern that allows us to extend the behavior of objects by placing these objects into a special wrapper class. The Decorator design pattern is quite popular in C# due to the fact that it helps us dynamically add behaviors to the wrapped objects.

The structure of this pattern consists of a Component class and a Concrete Component class from one part and a Decorator class and a Concrete Decorator class on the other side. The Concrete Decorator class is going to add additional behavior to our Concrete Component.

So, when do we use this pattern?

Well, we should use this pattern when we have a need to add additional behavior to our objects. Furthermore, we should use it when it is too complicated to use inheritance or when it doesn’t make sense at all (too many inherit layers, urge to modify existing inheritance hierarchy by adding some additional layers, etc.).

**Behavioral**

**Strategy**

The Strategy design pattern is a behavioral design pattern that allows us to define different functionalities, put each functionality in a separate class and make their objects interchangeable.

In other words, we have a main Context object that holds a reference towards a Strategy object and delegates it by executing its functionality. If we want to change the way the Context performs its work, we can just replace the current Strategy object with another one.

***Conclusion***

We should use this pattern whenever we have different variations for some functionality in an object and we want to switch from one variation to another in a runtime. Furthermore, if we have similar classes in our project that only differ on how they execute some behavior, the Strategy pattern should be the right choice for us.

We should consider introducing this pattern in situations where a single class has multiple conditions over different variations of the same functionality. That’s because the Strategy pattern lets us extract those variations into separate classes (concrete strategies). Then we can invoke them into the context class.

**Command**

The Command design pattern consists of the Invoker class, Command class/interface, Concrete command classes and the Receiver class.  Having that in mind, in our example, we are going to follow the same design structure.

***Conclusion***

Even though the Command design pattern introduces complexity to our code, it can be very useful.

With it, we can decouple classes that invoke operations from classes that perform these operations. Additionally, if we want to introduce new commands, we don’t have to modify existing classes. Instead, we can just add those new command classes to our project.