**Strategy** is a behavioural design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

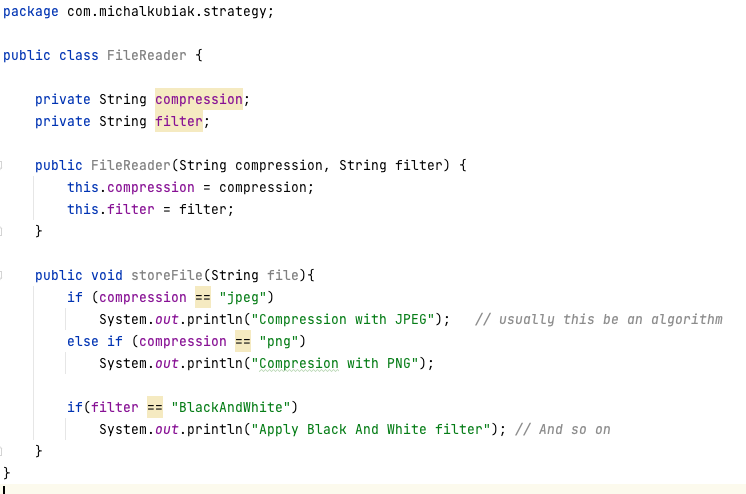
**The Pros** of Strategy are implementation details are isolated from the code that uses them, inheritance can be replaced with composition and Open Close Principle can introduce new strategies without changing the context. **The Cons** are that you clients need to be aware of the differences between strategies to be able to select a proper one, if you have a couple of algorithms that rarely change, then there is no reason to overcomplicate the code with new classes or interfaces that come along with the Strategy Pattern.

Let’s say in our application we want a user to store files or upload pictures. We create a new package under a new branch which we pushed to GitHub. The package contains a new class called FileReader.

The first thing we would usually do is compress that file. There are different types of image compression algorithms such a JPEG for example. Then, we can apply a filter so all of the images have the same look, for example a black and white filter.

We need a few private fields which determine the compression type, and the filter we will use. The first field will be a String called compression, it would be better to use an Enum but for simplicity I went with String, to demonstrate this Design Pattern. Another field was created called filter also of type String.

These fields can be initialized using a constructor, after which we can check the compression and filter types in the storeFile method. Next we can apply a filter, and system out print a corresponding message. Instead of the messages, in a real application we would use algorithms. This is a hypothetical class to demonstrate the initial problem and apply a **Strategy Design Pattern** to change the class to be more flexible.

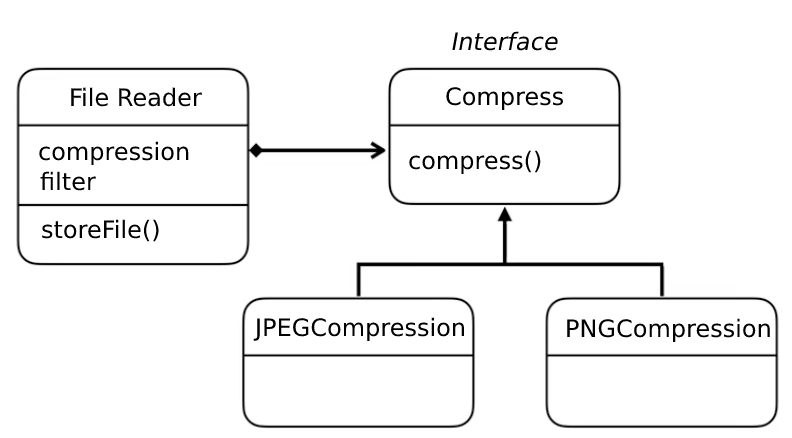


There are a number of problems with this code or this example of a class in an application. This class is violating the **Single Responsibility Principle** because with this implementation, because it is not only responsible for storing the file, it is also responsible for checking the compression type, and checking the filter or applying one. Instead of the SOUT we would have an algorithm there to compress a file, which would be a lot of lines of code. As this class gets bigger, it becomes unmaintainable. Which means we are violating the SRP principle.

The second issue, is that it would be hard to support new compression types or filters, we would have to add more if statements.

To solve these problems, we need to apply Polymorphism, we want our file reader to behave differently depending on compression or filter type we use. We can introduce a Compression interface with a method compress. Then we create classes which implement this interface such as JPEG and PNG compression classes. With this structure, we are following the **Open Close Principle.** If we want to add a new compression type, we just make a new class which implements the Compression interface. In out FileReader class we would change the compression type to Compressor, to maintain a reference to the compressor object.

The same problem would be used for the filtering, the BlackAndWhite would implement the Filter interface which would then be referenced as a type in the FileReader class. When we want the FileReader to maintain a reference to a Compress object we give it a concrete Compress object such as JPEGCompression or PNGCompression.



In the other case, the filtering would be the same, the Compress interface would be called Filter and it would have a method called applyFilter() which would be implemented by BlackAndWhite or any other class which dealt with filtering.

This is what we call the Strategy Pattern. The Gang of Four book has a structure of the Strategy Pattern while is the same as the one above, only the FileReader is ‘Context’, the Interface is the ‘Strategy’ and the JPEGCompression would be a ‘ConcreteStrategy’.

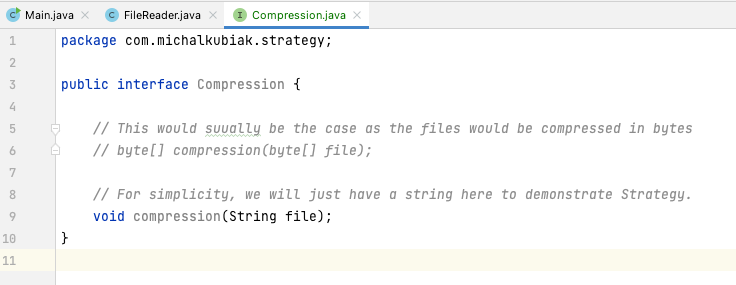
The context class must maintain a reference to a strategy object, the strategy is an interface *or* an abstract class that represents some sort of code or algorithm, and different classes implement it. It is similar to the State Pattern, both are structurally similar and both are used to change the behaviour of an object.

The difference between the State and Strategy Pattern is that in the State Pattern, the FileReader object can have a single state, and all the behaviours are represented by a subclass of the interface or abstract class it references. In the Strategy Pattern we do not have a single state, different behaviours are represented using different strategy objects. State would be considered an extension of Strategy, where both are based on composition but Strategy makes the helper objects independent, State does not restrict dependencies.

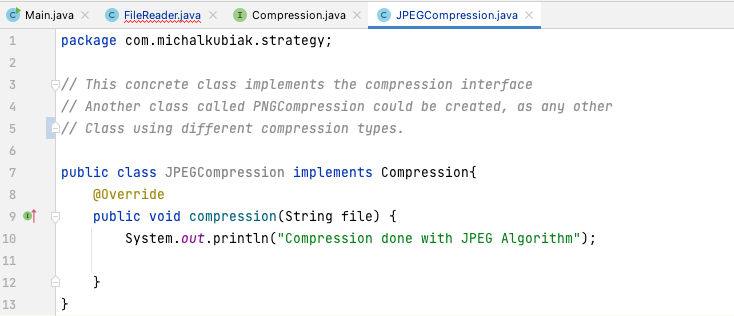
Strategy is used to isolate business logic of a class from the implementation details, it is used when a class has a lot of conditions or operations

Implementation:

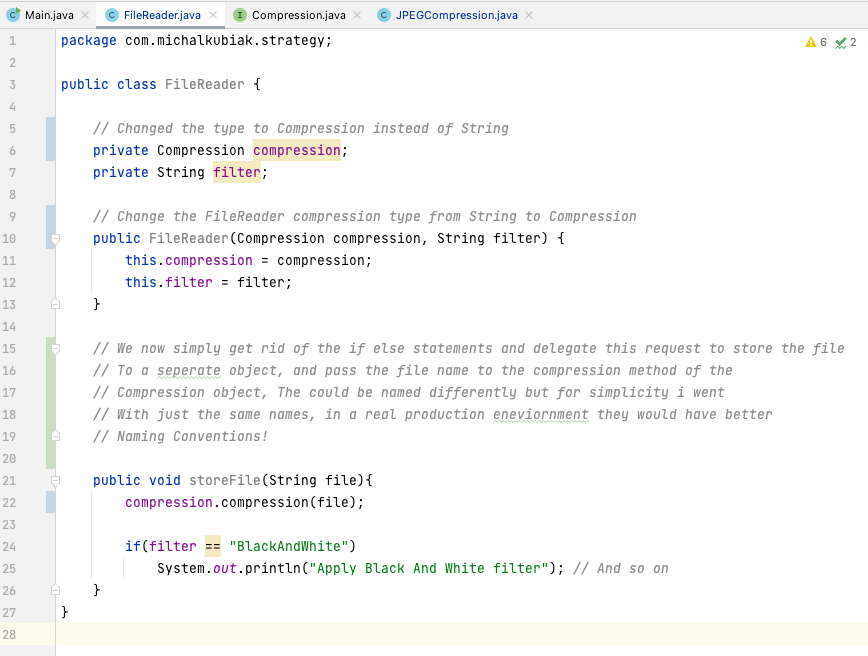
Compression Interface:



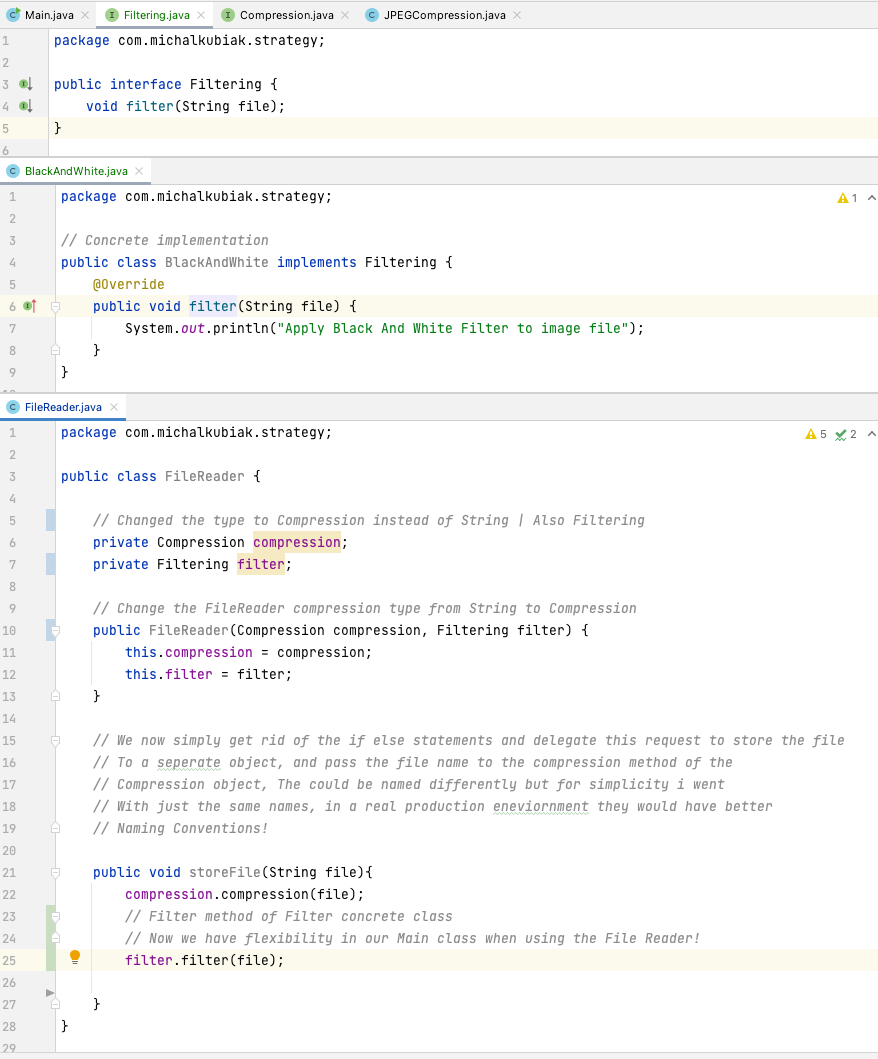
Concrete Compression Objects:



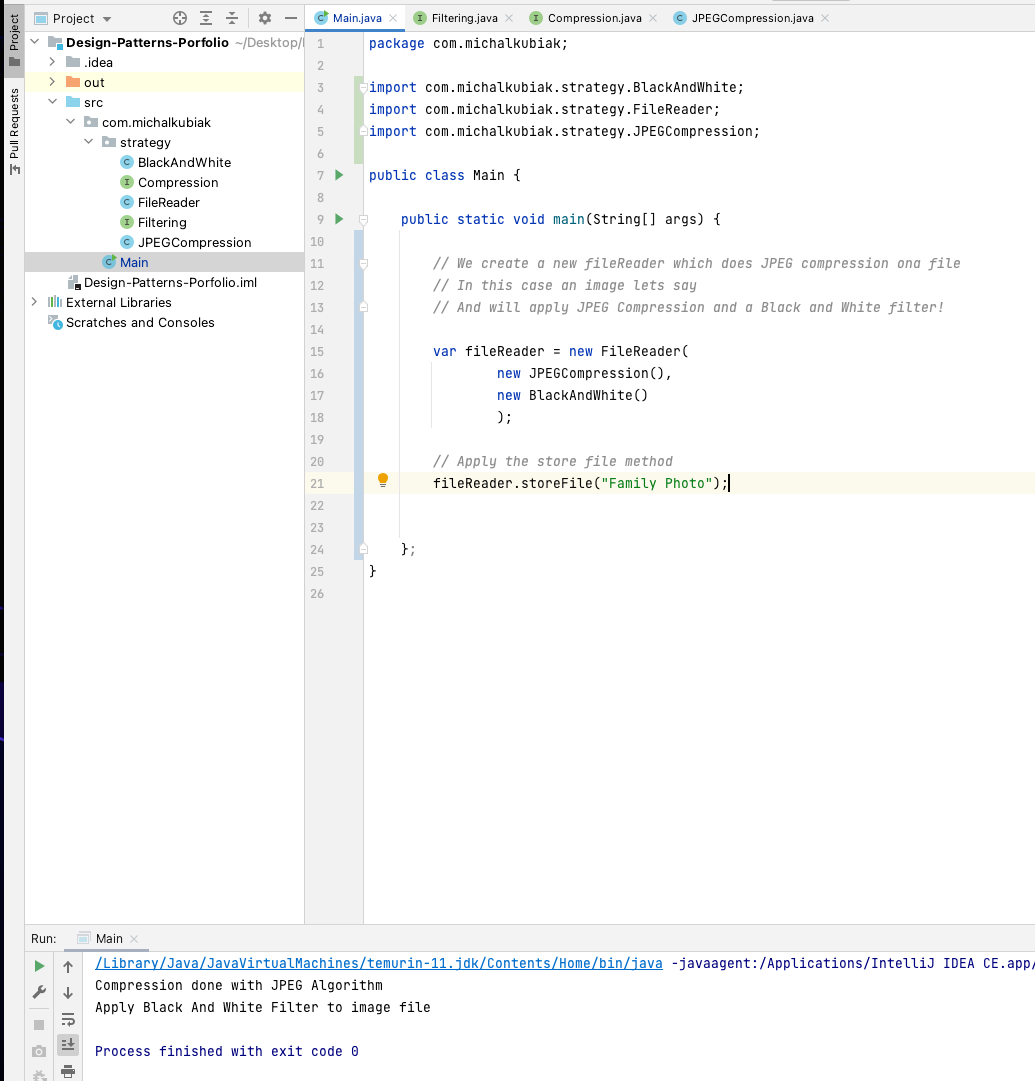
File Reader Type Changes for Compression Type:



Filtering Interface and subsequent filter class implementation and File Reader type changes:



Main Class Showcase and Use Cases:



Now when the app is compiled, we can see that the string ‘’Family Photo” has had a JPEG compression applied as well as it was filtered Black And White.

The application can now be extended by creating more filters or compression methods classes which implement their interfaces, making the app more maintainable and more flexible in the future!