This Factory class is responsible for creating and returning concrete implementations based on the algorithm of our choice (specified in the `web.config` file). This class is sealed, because it doesn't make sense to inherit it. Also, the constructor is private, so its instance cannot be created at all, and all methods are made static. Why static? Because `AlgorithmFactory` is behaving like a `Helper` class, helping the clients to get an instance of the appropriate `Algorithm implementation` class. See the static `GetSpecifiedAlgorithm()` method; it returns an instance of `IEncryptionAlgorithm`. This is where we see the important design principle—"Always program to interfaces". Because each encryption algorithm class implements `IEncryptionAlgorithm`, the supertype of each implementation is the same—`IEncryptionAlgorithm`. Hence, we don't need to worry about the actual implementation class as long as the class implements the same interface.

Step 5: Implement the Configuration Settings

Here is how we get the type specified in the `application config` file:

```
string algoType =
      System.Configuration.ConfigurationSettings.AppSettings["algo"];
```

In the `config` file, the following entry is made (under `appSettings`):

```
<add key="algo" value="NeekProtect.XOREncryption,NeekProtect"/>
```

In the value, we specify the fully qualified class name and the assembly name in which the class is present, separated by a comma. We need this entry as we will dynamically load the assembly using `Activator.CreateInstance`, as:

```
IEncryptionAlgorithm algoInstance;
algoInstance = Activator.CreateInstance(Type.GetType(algoTyp)) as
               IEncryptionAlgorithm;
```

If no entry is specified in the `config` file, then the `XOR Encryption` class is used as the default (see the code for `AlgorithmFactory`). `algoInstance` is then returned to the client caller.

Let us now see the client, and how it uses this instance. The class `EncryptionEngine` is our client here:

```
/// <summary>
/// This class prepares the byte array and sets files and directories
/// to be encrypted and calls the relevant encryption/decryption
/// algorithm
/// </summary>
    public class EncryptionEngine
{ ….}
```

This class has a method `EncryptFile`, as follows:

```
private void EncryptFile(string fullPath, string password)
{
               IEncryptionAlgorithm x =
                             AlgorithmFactory.GetSpecifiedAlgorithm();
                x.Password = password;
                x.KeySize = 128;
                string ext = Path.GetExtension(fullPath);
                byte[] extByte = Encoding.ASCII.GetBytes(ext);
                byte[] inputByte = File.ReadAllBytes(fullPath);
                MemoryStream m = new MemoryStream();
                m.Write(inputByte, 0, inputByte.Length);
                m.Write(extByte, 0, extByte.Length);
                x.RawInput = m.ToArray();
                byte[] o = x.Encrypt();
                …..//other stuff
}
```

We call the `AlgorithmFactory`'s static method `GetSpecifiedAlgorithm()`, which returns an instance of the class specified in the `config` file. Here, we don't care how the instance is returned; we just need an instance of `IEncryptionAlgorithm` so that we can encrypt the input file. This is another example of programming to interfaces.

Step 6: Implement another Custom Algorithm

In the above code, there is no mention of the two encryption classes we have coded earlier: `XOREncryption` and `RijndaelEncryption`. We only use `IEncryptionAlgorithm` in this code, and this gives us the flexibility to use any custom algorithm as long as it implements `IEncryptionAlgorithm`.

Let's suppose a user installs our encryption program, and is not satisfied by the two default algorithms provided. Now he or she wishes to use the `3DES` algorithm with the program. How can he or she do that?

The user will need to create their own `3DESEncryption` class and implement all of the methods and properties specified in the `IEncryptionAlgorithm` interface, as follows:

```
public class 3DESEncryption: IEncryptionAlgorithm
{
       //implement IEncryptionAlgorithm properties and methods
}
```

Now the user has to compile it and place the resultant assembly in the folder where our program is installed, which is probably something like this: `C:\Program Files\PP\Encryption Program`.