In the same folder, the user has to open and edit the XML `configuration` file (usually this file is named as `App.GUI.exe.config`) and create/overwrite the value of the key `algo` as:

```
<add key="algo" value="MyCustomAlgo.3DESEncryption, MyCustomAlgo "/>
```

(For the sake of illustration, we have assumed the namespace to be `MyCustomAlgo`).

Now the Factory class will return this instance to be used in our encryption program. So we see how the "plugin" type functionality can be achieved by using the Dependency Injection, and users can build their own custom implementations and plug them into our program.

Command Design Pattern

The Command design pattern is used to decouple the calling object and the object that is being called, so that the calling object does not know how its request will be executed by the called object. Let us understand this with the help of an example.

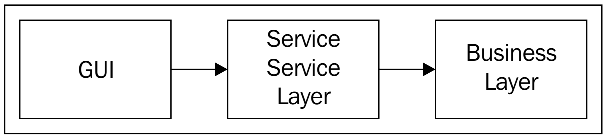
Usually, in association or aggregation relationships, the object holding a reference to another object will communicate with this object by invoking a method on it. For example, in our OMS web application, the `AddEditCustomer.aspx.cs` page object has a reference to the `Customer` object (an association, as the `Customer` object is used locally). The page object loads a particular `Customer` object by calling a method on it, for example:

```
Customer customer = new Customer();
customer.Load(customerID);
```

So the page object is interacting with the customer object by issuing a command to it (`Load()`, in our case). This is how objects usually interact and communicate with each other—by invoking commands.

Decoupling the GUI completely from the BL

Now let's take another scenario. In Chapter 4, we saw how we can use a 5-tier configuration and introduce loose coupling into our projects. In the sample OMS code, we saw that the GUI and the BL were loosely coupled, but the GUI was dependent on the BL in the sense that we had a reference to the BL in our GUI code (using statements referring to the BL Tier). What if we want the BL and the GUI to be completely independent of each other, where there is not even a one way reference between the GUI and the BL. This means that the GUI will not call business layer objects directly, as it doesn't know anything about them. One way to make this happen is to add another level of indirection by adding another tier. We can call it the Service Layer. The GUI can refer to and call this new Service layer method, which sits in between the GUI and the Business Layer and handles the abstraction. Here, the GUI talks to the Service Layer, and the Service Layer in turn interacts with the Business Layer.



This decoupling will make our application more flexible as the GUI has no reference to the BL tier, and it does not need to know how the business logic will be handled. We can have multiple GUIs for the same business logic, and multiple business logic implementations for the same GUI. This complete independence makes software development much more flexible and robust, but also increases complexity.

In this scenario, assume that we are creating a GUI form to add a new customer. In the `Add` `buttons click` event handler, we cannot simply create a new `Customer` object because the GUI has no reference of the `Customer` class in the Business Layer. Additionally, it does not know which object would be there, and what methods that object would support. In such circumstances, we use the Command design pattern to abstract the invocation of the commands to associated objects.

Creating the Command Interface

In this design pattern, we create a `Command` object, which will encapsulate the actual command. Then the GUI will invoke this `command` object, which will issue the actual command. So a GUI object may be referred to as an "invoker" here, and the service layer would be the "receiver". First, we define an interface `ICommand`, and every concrete command implementation will implement this interface. This command infrastructure will help us decouple the GUI and the BL.

The `ICommand` interface will have only one method, `Execute()`, as follows:

```
public interface ICommand
    {
        CommandResult Execute(CommandArg cmdArg);
    }
```