```
      {
   product  = factory.CreateProduct(productType);
   //set product properties
   product.OrderDate = Datetime.Now;
   //reset the product count in the inventory as this product has been
   //ordered
   product.ResetInventory();
   return product;
      }
   }
```

In the above code, the `ProductFactory` class has abstracted the product creation logic. So if there are new products to be added in future, we don't need to change the code in the `ProductManager` class; only the `factory` class needs to be changed. We have de-coupled the `ProductManager` class and made it more flexible by adding another level of indirection in terms of a `factory` class.

Another example where you will find factory design useful is when you want your application to be able to talk to different database types. You may not know which database each of your clients (to whom you sold your application) has. It could be MS SQL Server, Oracle, FireBird, PostGres, or MySql.

You have developed the data access layer classes for each of these databases in your application code. Now you want your application to be able to instantiate any of the DAL classes based on the actual customer database. Our application does not know beforehand which database will be used. For such cases, and similar scenarios, the factory design pattern can help.

We will see another slightly different, real world, practical example of a Factory design in the next design pattern, **Dependency Injection** (**DI**).

Dependency Injection

The Dependency Injection and factory design patterns are very common, and provide great flexibility in software development. Although most programmers have come across these patterns, they may not grasp the concepts completely until they see these patterns in action in real projects.

In this section, we will learn how to achieve loose coupling and "plug-and-play" architecture using these patterns, with the help of a sample project—a flexible encryption program. We will be focusing on the code from the viewpoint of understanding the Dependency Injection design pattern. Therefore, the detailed syntax and complete code will not be listed here. A complete working code for this example is provided in the code bundle.

The Dependency Injection (DI) design pattern is "a form of" the **Inversion of Control** (**IoC**) design which is applied in many frameworks. DI gives the flexibility of attaching a custom implementation such as a "plugin", without modifying existing software. Dependency Injection can be achieved using Constructor, Setter, or Interface Injection. In this chapter, we will learn and understand Interface Injection, which is quite common and more flexible than the other two approaches.

Basic Approach

We will follow a set of steps to achieve Dependency Injection in our working sample, as described below.

Step 1: Create an Interface

Let us start with our encryption program by authoring an interface so that others can implement their own algorithmic implementations by defining these methods in their own custom way:

```
public interface IEncryptionAlgorithm
{
    string Password
    {
        get;
        set;
    }
    byte[] RawInput
    {
        get;
        set;
    }

    byte[] Salt
    {
        set;
    }

    int KeySize
    {
        set;
    }
    byte[] Encrypt();
    byte[] Decrypt();
    bool CheckPassword();
}
```