Now, because we have used `IProduct` in the method signature, we can pass in all current and future concrete implementation classes in this method as long as they implement the `IProduct` interface:

```
IProduct  p = new EletronicProduct();
return p;
```

Also, programming to interfaces gives us the flexibility of **Dynamic polymorphism**. We don't have to worry about the type of concrete implementations, as they are all deriving from a common abstract class or implementing a common interface. There is something we avoid—writing multiple `Create` methods for different product types.

The Need for Factory Design

So far so good. But we still need to make sure that our `OrderProduct` method can return multiple product types. How does the code in `OrderProduct()` method know which concrete type to return? One option is that we modify the code so that it can include the type of the product passed as a second parameter. Then, by using a `switch` case or `if else` statements, we can have different creational logic implemented for each concrete type, as follows:

```
public IProduct OrderProduct(string productType)
  {
     IProduct product;
     switch(productType)
   {
  case "BeautyProduct": product = new BeautyProduct();
                        break;
  case "ElectronicProduct": product = new ElectronicProduct ();
                        break;
  case "FoodProduct": product = new FoodProduct ();
                        break;
  default: break;
   }
//set product properties
product.OrderDate = Datetime.Now;
//reset the product count in the inventory as this product has been
//ordered
product.ResetInventory();

 return product;
  }
```

As you can see, we now face another issue. If we add a new product in future, we will still need to modify our `OrderProduct` method to add another `switch case` statement for the newly-added product, so that `OrderProduct` can return it.

To fix this issue, and to make our code more elegant and flexible, we need to abstract the product creation logic to some other class. This is where the factory design can help us.

We take the product creation code out of the `ProductManager` class and create a new class to handle and return the correct type of product object. So the only role of this new class would be to create and return product objects based on the object type passed to it. This will make our `ProductManager` class independent of handling the correct product type and allow it to focus on other important business rules that may need to be applied for managing products.

We will call this class `ProductFactory`. Here is the code for that class:

```
public class ProductFactory
{
  public IProduct CreateProduct(string productType)
  {
     IProduct product = null;
     switch(productType)
   {
  case "BeautyProduct": product = new BeautyProduct();
                        break;
  case "ElectronicProduct": product = new ElectronicProduct ();
                        break;
  case "FoodProduct": product = new FoodProduct ();
                        break;
  default: break;
   }
 return product;
  }
}
```

This `factory` class takes the product type as a string parameter and simply returns the appropriate concrete product object based on this parameter. Here is how our `ProductManager` will use this `factory` class:

```
public class ProductManager
{
ProductFactory factory= new ProductFactory();
IProduct product;
 //misc. methods to handle products
public IProduct OrderProduct(string productType)
```