The problem with this method of using an object is that there is no flexibility, and the objects are tightly coupled with their class types. For example, we might need to use the `Product` object in different cases, as we might have a `ProductManager` class that will create and return a product to be added in an `OrderLineItem` object (this class will return an appropriate product object based on the type of product a customer has ordered). The following is a sample code of such a `ProductManager` class that returns a `BeautyProduct` object:

```
public class ProductManager
{
 //misc. methods to handle products
//
public BeautyProduct OrderProduct()
  {
BeautyProduct bp = new BeautyProduct();
//set bp properties
bp.OrderDate = Datetime.Now;
//reset the product count in the inventory as this product has been
//ordered
bp.ResetInventory();
return bp;
   }
}
```

In the above code sample, we have the `OrderProduct()` method that creates a new instance of `BeautyProduct`, and returns it to the consuming class (which might use this object to perform operations on it, as in an `Order` class).

We will not be focusing on other methods in the `Product/ ProductManager` class that are related to handling and applying business logic related methods and properties, as our main focus is to understand the approach to solve problems in a better way by using design patterns!

The problem with this way of programming is that it makes our application too rigid, because to create another type of product object, say `ElectronicProduct`, we need to write another `OrderProduct` method in the `ProductManager` class. The `ProductManager` class returns an object of type, `ElectronicProduct`, which looks something like this:

```
public ElectronicProduct OrderEletronicProduct()
  {
ElectronicProduct ep = new ElectronicProduct ();
//set ep properties
ep.OrderDate = Datetime.Now;
```

```
//reset the product count in the inventory as this product has been
//ordered
ep.ResetInventory();
return ep;
}
```

So for each type of product, we need to add multiple order methods with different return signatures, which is quite messy. Also, in future, if we add a new type of product, we need to open our `ProductManager` class and modify it to make sure it can create and return the new product type. So this approach makes our code very rigid, inflexible and open to modifications each time there is a change. To avoid this, we can use **Polymorphism** and **program to interfaces** instead of using the concrete classes.

So what does 'programming to interfaces' mean? We create an object using the interface/super class as the type instead of the concrete classes. This means:

```
BeautyProduct bp = new BeautyProduct();
```

becomes

```
IProduct bp = new BeautyProduct();
```

Note that the type of the `bp` object is now the interface, making it possible for us to switch to different concrete types during code execution. This is known as **Runtime polymorphism**. Let us see how we can use this approach to make our `ProductManager` class better:

```
public class ProductManager
{

public IProduct OrderProduct()
  {

//misc. methods
IProduct bp = new BeautyProduct();
//set bp properties
bp.OrderDate = Datetime.Now;
//reset the product count in the inventory as this product has been
//ordered
bp.ResetInventory();
       return bp;
   }
}
```