**Provide an interface for creating families of related or dependent objects without specifying their concrete classes**

**Understanding and Implementing Abstract Factory Pattern in C#**

[**http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa**](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

By [Rahul Rajat Singh](http://www.codeproject.com/script/Membership/View.aspx?mid=3402606), 10 Feb 2012

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | |  |

Top of Form

* [Download source - 6.72 KB](http://www.codeproject.com/KB/architecture/328373/AbstractFactoryTest.zip)

**Introduction**

This article aims at understanding and implementing Abstract Factory Pattern in C#.

**Background**

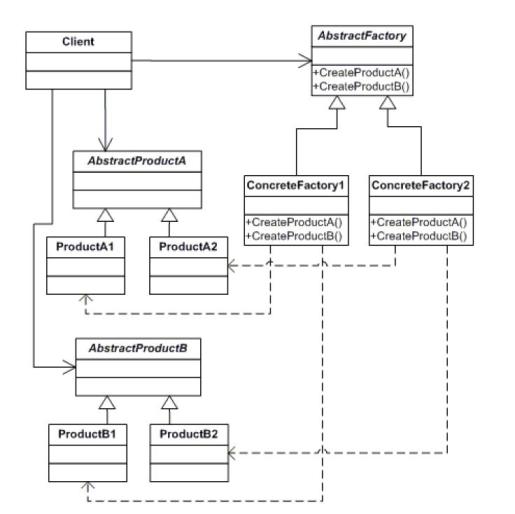
Abstract factory pattern in useful when the client needs to create objects which are somehow related. If we need to create the family of related or dependent objects, then we can use Abstract Factory Pattern.

This pattern is particularly useful when the client doesn't know exactly what type to create. As an example, let's say a Showroom exclusively selling cellphones gets a query for the smart phones made by Samsung. Here we don't know the exact type of object to be created (assuming all the information for a phone is wrapped in the form of a concrete object). But we do know that we are looking for smart phones that are manufactured by Samsung. This information can actually be utilized if our design has Abstract factory implementation.

So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects. We will go ahead and write a rudimentary application for the scenario we just talked about.

**Using the Code**

Let us start with the GOFs representation of Abstract Factory Pattern:



Let's see what each class does here:

* AbstractFactory: declares an interface for operations that create abstract products
* ConcreteFactory: implements the operations to create concrete product objects
* AbstractProduct: declares an interface for a type of product object
* Product: defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface
* Client: uses interfaces declared by AbstractFactory and AbstractProduct classes

Now let us focus on the problem at hand. We need to create the appropriate object containing the information about cell phone based on the user request of 1. Type of phone 2. Phone manufacturer. For the sake of simplicity, let's assume we have 3 manufacturers:

1. Nokia
2. Samsung
3. HTC

and there could be two types of phones:

1. Smart Phones
2. Dumb Phones

So with this information, we can safely say that we need three concrete factories (one for each manufacturer) and two sets of related products (one for smart and one for dumb).

**Creating the Abstract Products**

In our case, we need two abstract products ISmart and IDumb.

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

interface IDumb

{

string Name();

}

interface ISmart

{

string Name();

}

**Creating the Concrete Products**

Now let us go ahead and create some concrete products for IDumb:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

class Asha : IDumb

{

public string Name()

{

return "Asha";

}

}

class Primo : IDumb

{

public string Name()

{

return "Guru";

}

}

class Genie : IDumb

{

public string Name()

{

return "Genie";

}

}

Let's do the same for ISmart:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

class Lumia : ISmart

{

public string Name()

{

return "Lumia";

}

}

class GalaxyS2 : ISmart

{

public string Name()

{

return "GalaxyS2";

}

}

class Titan : ISmart

{

public string Name()

{

return "Titan";

}

}

So we have all the concrete classes ready for all the Dumb Phones and smart phones irrespective of their manufacturers.

**Creating the Abstract Factory**

Now the way we associate these Concrete products with their manufacturers is using the Concrete factories. But before having the concrete factories, we need to have an Abstract Factory.

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

interface IPhoneFactory //'I' stands for interface no relation with Iphone

{

ISmart GetSmart();

IDumb GetDumb();

}

**Creating the Concrete Factories**

Now we can create our Concrete Factories for each manufacturer:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

class SamsungFactory : IPhoneFactory

{

public ISmart GetSmart()

{

return new GalaxyS2();

}

public IDumb GetDumb()

{

return new Primo();

}

}

class HTCFactory : IPhoneFactory

{

public ISmart GetSmart()

{

return new Titan();

}

public IDumb GetDumb()

{

return new Genie();

}

}

class NokiaFactory : IPhoneFactory

{

public ISmart GetSmart()

{

return new Lumia();

}

public IDumb GetDumb()

{

return new Asha();

}

}

**Creating the Client**

Now we have all the Abstract product classes ready, all the Concrete Product classes ready. Our Abstract Factory is ready and all the Concrete Factories are ready. Now we can write client that will use this hierarchy of related products to create the products.

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/328373/Understanding-and-Implementing-Abstract-Factory-Pa)

enum MANUFACTURERS

{

SAMSUNG,

HTC,

NOKIA

}

class PhoneTypeChecker

{

ISmart sam;

IDumb htc;

IPhoneFactory factory;

MANUFACTURERS manu;

public PhoneTypeChecker(MANUFACTURERS m)

{

manu = m;

}

public void CheckProducts()

{

switch (manu)

{

case MANUFACTURERS.SAMSUNG:

factory = new SamsungFactory();

break;

case MANUFACTURERS.HTC:

factory = new HTCFactory();

break;

case MANUFACTURERS.NOKIA:

factory = new NokiaFactory();

break;

}

Console.WriteLine(manu.ToString() + ":\nSmart Phone: " +

factory.GetSmart().Name() + "\nDumb Phone: " + factory.GetDumb().Name());

}

}

static void Main(string[] args)

{

PhoneTypeChecker checker = new PhoneTypeChecker(MANUFACTURERS.SAMSUNG);

checker.CheckProducts();

Console.ReadLine();

checker = new PhoneTypeChecker(MANUFACTURERS.HTC);

checker.CheckProducts();

Console.ReadLine();

checker = new PhoneTypeChecker(MANUFACTURERS.NOKIA);

checker.CheckProducts();

Console.Read();

}

Now we can say we have a basic skeleton for the Abstract factory pattern ready. The concrete products here are not telling anything but names of products but they can contain more information too. Before we end the show, we can have a class diagram for the classes we created so that we can use this to map it with the GOFs diagram.



**Note**: Please refer to the source code for implementation. Stepping through the code will really help in understanding this concept better.

Bottom of Form