**Motivations:**

Suppose we want information regarding electronics devices used in Bangladesh. And we want the information based on country the devices are made of. Like Indian, Chinese and Bangladeshi. Here Indian, Chinese and Bangladeshi are the families of electronic devices (from the definition we know that AbstractFactory pattern creates families of related objects) and electronic devices like TV and Fridge are the products.

Or we may want to know about the population of Bangladesh. We want the information based on the division (region) like Dhaka, Chittagong and Rajshahi. Here Dhaka, Chittagong and Rajshahi are the families of population and people like men and women may be the products.

Or we may want to work with Database objects. We also want to work with the object of different types of Database system like MS SQL, MySQL or Oracle. Here MS SQL, MySQL and Oracle are the families of database object. And database objects like connection and transaction are the products.

These types of problem can be solved using AbstractFactory pattern.

**Definition**

The original definition from the book of GoF is**” Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”** Let’s elaborate the definition:

1. There will be an abstract base class (not an implementation) known as AbstractFactory. It will hold the operations (abstract methods) to create the abstract (not concrete) products.
2. This base AbstractFactory class will have one or more implementations known as ConcreteFactory. It implements the operations (from the AbstractFactory) to create concrete products of certain family.
3. The Client will deal only with abstract type. The concrete type will be created by the factory only.
4. Consumer or main method will use the products through Client class.

So participants of Abstract Factory patterns are:

1. AbstractFactory
2. AbstractProduct
3. ConcreteFactory
4. ConcreteProduct and
5. Client

Structure of the participants (using c# codes) are as follows:

* AbstractFactory:

abstract class AbstractFactory

  {

    public abstract AbstractProductA CreateProductA();

public abstract AbstractProductB CreateProductB();

//there may be more products like

// public abstract AbstractProduct1 CreateProduct1();

// public abstract AbstractProduct2 CreateProduct2();

// public abstract AbstractProduct3 CreateProduct3();

  }

* ConcreteFactory:

class ConcreteFactory1 : AbstractFactory

  {

    public override AbstractProductA CreateProductA()

    {

      return new ProductA1();

    }

public override AbstractProductB CreateProductB()

    {

      return new ProductB1();

    }

  }

class ConcreteFactory2 : AbstractFactory

  {

    public override AbstractProductA CreateProductA()

    {

      return new ProductA2();

    }

    public override AbstractProductB CreateProductB()

    {

      return new ProductB2();

    }

  }

* AbstractProduct:

abstract class AbstractProduct

  {

    public abstract void DoSomthing();

  }

* ConcreteProduct:

 class ProductA1 : AbstractProduct

  {

    public override void DoSomthing ()

    {

    }

  }

 class ProductB1 : AbstractProduct

  {

    public override void DoSomthing ()

    {

    }

  }

 class ProductA2 : AbstractProduct

  {

    public override void DoSomthing ()

    {

    }

  }

 class ProductB2 : AbstractProduct

  {

    public override void DoSomthing ()

    {

    }

  }

And following is the collaboration between participants:

1. class MainApp
2. {
3. public static void Main()
4. {
5. AbstractFactory factory = new ConcreteFactory1();
6. Or factory can be initialized by ConcreteFactory2() like
7. AbstractFactory factory = new ConcreteFactory2();
8. Or there can be many more factories. We can even initialize
9. them using another design pattern called FactoryMethod.
10. Client client = new Client(factory);
11. client.DoSomething();
12. }
13. }

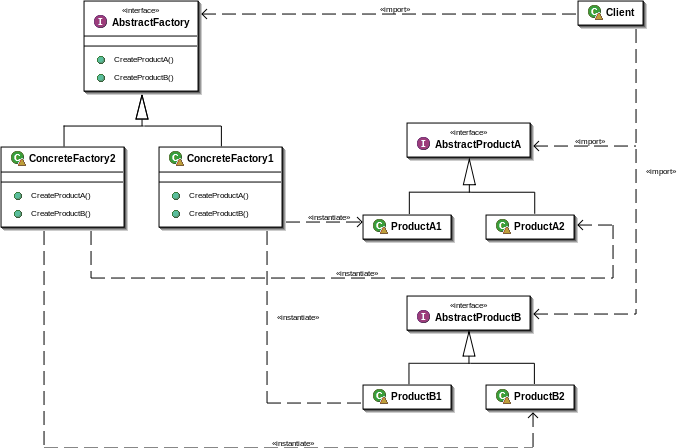
At line no 5, we have our first participant AbstractFactory which is implemented by ConcreteFactory1 (another participant). There may be multiple ConcreteFactory. At line no 14, we have our participant Client. Client will take the implementation of the factory as a constructor parameter and create the concrete product based on the parameter. At line no 15, we call a method of Client. The method will use the concrete product created at the constructor.

Let’s see the structure of Client In detail:

1. class Client
2. {
3. private AbstractProduct \_abstractProductA;
4. private AbstractProduct \_abstractProductB;
5. public Client(AbstractFactory factory)
6. {
7. \_abstractProductA = factory.CreateProductA();
8. \_abstractProductB = factory.CreateProductB();
10. If factory is ConcreteFactory1 then
11. \_abstractProductA will be ProductA1 and
12. \_abstractProductB will be ProductB1
14. If factory is ConcreteFactory2 then
15. \_abstractProductA will be ProductA2 and
16. \_abstractProductB will be ProductB2
17. }
18. public void DoSomething()
19. {
20. Do something with \_abstractProductA or \_abstractProductB;
21. }
22. }

At line no 3, we have another participant AbstractProduct. At line no 6, we have constructor function that takes factory as a parameter. Using this parameter ConcreteProduct (another participant) will be created. See at line no 8 and 9.

Following is the graphical representation of above code:



Let us implement the first practical example we have discussed at the motivation section or if you forgot following was the example: Suppose we want information regarding electronics devices used in Bangladesh. And we want the information based on country the devices are made of. Like Indian, Chinese and Bangladeshi. Here Indian, Chinese and Bangladeshi are the families of electronic devices (from the definition we know that AbstractFactory pattern creates families of related objects) and electronic devices like TV and Fridge are the products.

Following is the C# implementation of above problem. Just copy and run it in your console application:

namespace Patttern.Examples.AbstractFactory

{

public class Program

{

//Consumer of AbstructFactory

static void Main(string[] args)

{

ElectronicsFactory factory = new IndianFactory();

//ElectronicsFactory factory = new ChineseFactory();

//there can be many more factories. We can even initialize

//them using another design pattern called FactoryMethod.

Client client = new Client(factory);// passing the factory to the client

client.DisplayProducts();

// Wait for user input

Console.ReadKey();

// If we initiallze the factory with IndianFactory, then the result will be

//IndianTV

//IndianFridge

// If we initiallze the factory with ChineseFactory, then the result will be

//ChineseTV

//ChineseFridge

}

}

// Client

public class Client {

ElectronicsProduct \_productTV;

ElectronicsProduct \_productFridge;

public Client(ElectronicsFactory factory) {

\_productTV = factory.CreatProductTV();

\_productFridge = factory.CreatProductFridge();

//If the factory is IndianFactory then

//\_productTV will be IndianTV and

//\_productFridge will be IndianFridge

//If the factory is ChineseFactory then

//\_productTV will be ChineseTV and

//\_productFridge will be ChineseFridge

}

public void DisplayProducts() {

\_productTV.Display();

\_productFridge.Display();

}

}

//AbstractFactory class

public abstract class ElectronicsFactory {

public abstract ElectronicsProduct CreatProductTV();

public abstract ElectronicsProduct CreatProductFridge();

}

//ConcreteFactory class

public class IndianFactory : ElectronicsFactory {

//Methode returns ConcreteProduct

public override ElectronicsProduct CreatProductTV()

{

return new IndianTV();

}

//Methode returns ConcreteProduct

public override ElectronicsProduct CreatProductFridge()

{

return new IndianFridge();

}

}

//ConcreteFactory class

public class ChineseFactory : ElectronicsFactory

{

//Methode returns ConcreteProduct

public override ElectronicsProduct CreatProductTV()

{

return new ChineseTV();

}

//Methode returns ConcreteProduct

public override ElectronicsProduct CreatProductFridge()

{

return new ChineseFridge();

}

}

//AbstractProduct class

public abstract class ElectronicsProduct

{

public abstract void Display();

}

//ConcreteProduct class

public class IndianTV : ElectronicsProduct {

public override void Display()

{

Console.WriteLine(this.GetType().Name );

}

}

//ConcreteProduct class

public class IndianFridge : ElectronicsProduct

{

public override void Display()

{

Console.WriteLine(this.GetType().Name);

}

}

//ConcreteProduct class

public class ChineseTV : ElectronicsProduct

{

public override void Display()

{

Console.WriteLine(this.GetType().Name);

}

}

//ConcreteProduct class

public class ChineseFridge : ElectronicsProduct

{

public override void Display()

{

Console.WriteLine(this.GetType().Name);

}

}

}