Порождающие паттерны

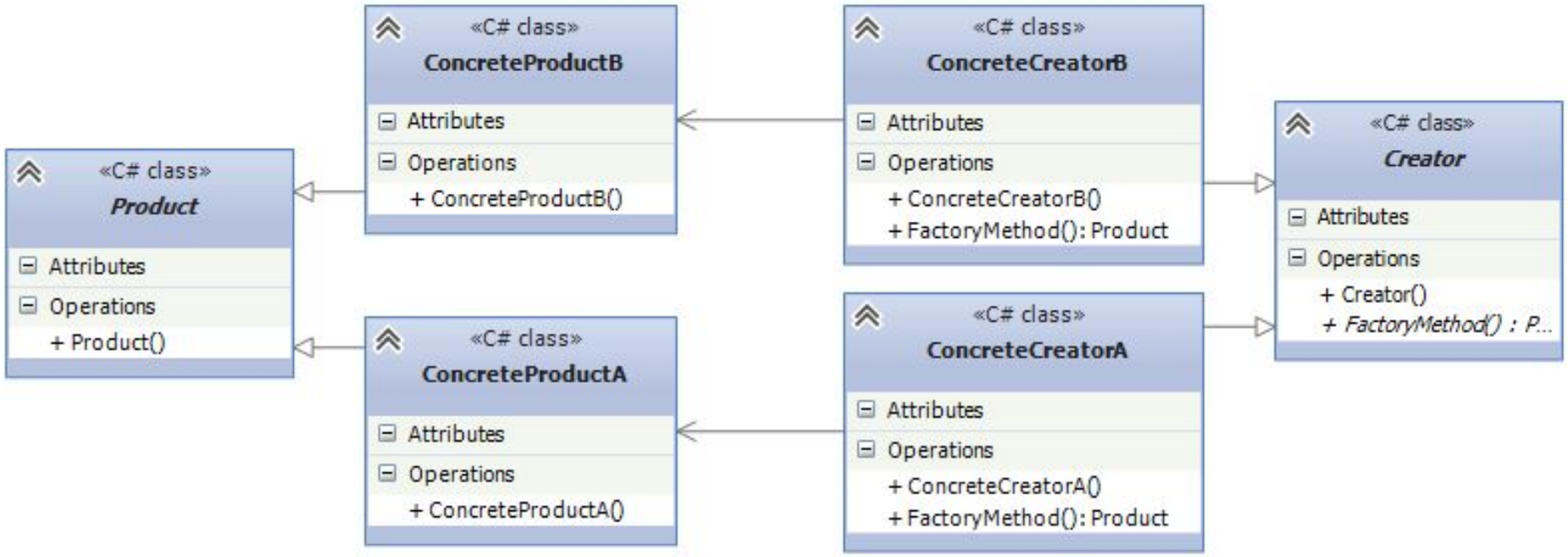
Фабричный метод (Factory Method)

- это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах.

Когда надо применять паттерн

- ✔ Когда заранее неизвестно, объекты каких типов необходимо создавать
- ✔ Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
- ✔ Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

UML – диаграмма паттерна «Factory Method»



Формальное определение паттерна на языке C#

```
abstract class Product    {}

class ConcreteProductA : Product   {}

class ConcreteProductB : Product   {}

abstract class Creator
{
    public abstract Product FactoryMethod();
}

class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod()
      { return new ConcreteProductA(); }
}

class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod()
      { return new ConcreteProductB(); }
}
```

Пример:

```
class Program
{
   static void Main(string[] args)
   {
      Developer dev = new
        PanelDeveloper("ООО КирпичСтрой");
      House house2 = dev.Create();
      dev = new WoodDeveloper("Частный
                                застройщик");
      House house = dev.Create();
       Console.ReadLine();
   }
}
// абстрактный класс строй-компании
abstract class Developer
{
   public string Name { get; set; }
    public Developer (string n) { Name = n; }
   // фабричный метод
   abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
   public PanelDeveloper(string n) : base(n){ }
    public override House Create()
   {  return new PanelHouse();}
}
// строит деревянные дома
class WoodDeveloper : Developer
{
   public WoodDeveloper(string n) : base(n){ }
  public override House Create()
   {  return new WoodHouse();  }
}

 abstract class House{ }

 class PanelHouse : House
{
   public PanelHouse()
   { Console.WriteLine("Панельный дом построен");  }
}

class WoodHouse : House
{
   public WoodHouse()
   { Console.WriteLine("Деревянный дом построен");  }
}
```

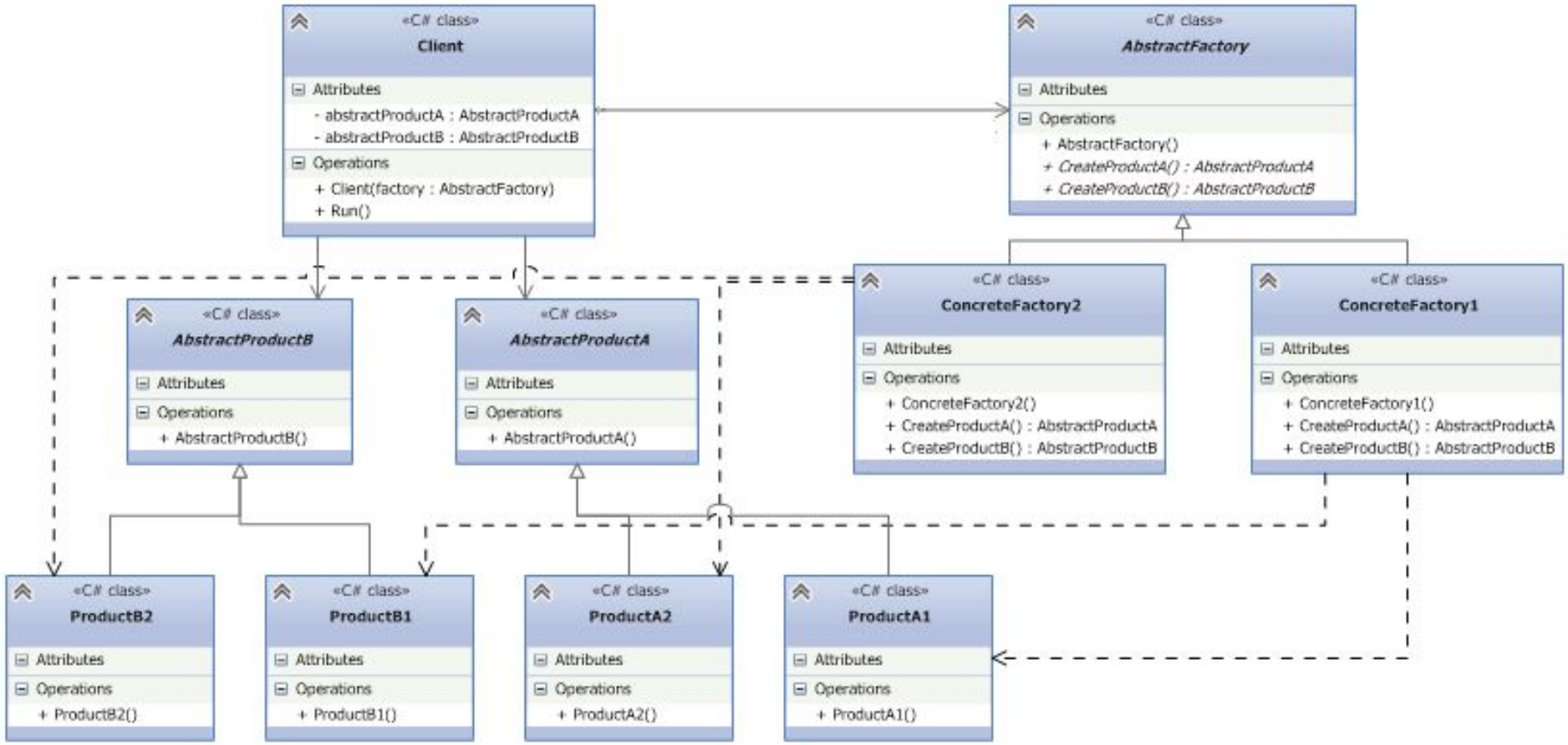
Абстрактная фабрика (Abstract Factory)

предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

Когда надо применять паттерн

- ✔ Когда система не должна зависеть от способа создания и компоновки новых объектов
- ✔ Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

UML – диаграмма паттерна «Abstract Factory»



Формальное определение паттерна на языке C#

```
abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
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(); }
}
 abstract class AbstractProductA  {}
 abstract class AbstractProductB   {}
 class ProductA1: AbstractProductA {}
 class ProductB1: AbstractProductB  {}
 class ProductA2: AbstractProductA  {}
 class ProductB2: AbstractProductB  {}
 class Client
{
    private AbstractProductA abstractProductA;
    private AbstractProductB abstractProductB;
     public Client(AbstractFactory factory)
    {       abstractProductB = factory.CreateProductB();    abstractProductA = factory.CreateProductA();    }
    public void Run()    { }
}
```

Пример:

```
class Program
{
   static void Main(string[] args)
   {
      Hero elf = new Hero(new ElfFactory());
      elf.Hit();          elf.Run();
      Hero voin = new Hero(new VoinFactory());
      voin.Hit();       voin.Run();
   }
}
abstract class Weapon//абстрактный класс – оружие
{   public abstract void Hit();   }
abstract class Movement// абстрактный класс движение
{   public abstract void Move();    }
class Arbalet : Weapon // класс арбалет
{
   public override void Hit()
   {    Console.WriteLine("Стреляем из арбалета");   }
}
class Sword : Weapon// класс меч
{
   public override void Hit()
   {     Console.WriteLine("Бьем мечом");    }
}
class FlyMovement : Movement// движение полета
{
   public override void Move()
   {   Console.WriteLine("Летим");  }
}
class RunMovement : Movement// движение – бег
{
   public override void Move()
   {    Console.WriteLine("Бежим");   }
}
abstract class HeroFactory// класс абстрактной фабрики
{
   public abstract Movement CreateMovement();
   public abstract Weapon CreateWeapon();
}
// Фабрика создания летящего героя с арбалетом
class ElfFactory : HeroFactory
{
   public override Movement CreateMovement()
   {    return new FlyMovement();   }
    public override Weapon CreateWeapon()
   {    return new Arbalet();     }
}
// Фабрика создания бегущего героя с мечом
class VoinFactory : HeroFactory
{
   public override Movement CreateMovement()
   {    return new RunMovement();    }
    public override Weapon CreateWeapon()
   {    return new Sword();    }
}
class Hero  // клиент - сам супергерой
{
   private Weapon weapon;
   private Movement movement;
   public Hero(HeroFactory factory)
   {
      weapon = factory.CreateWeapon();
      movement = factory.CreateMovement();
   }
   public void Run()   {       movement.Move();}
   public void Hit()   {   weapon.Hit();    }
}
```

Одиночка (Singleton)

предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

Когда надо применять паттерн

- ✔ Когда система не должна зависеть от способа создания и компоновки новых объектов
- ✔ Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными