W4 V

Фабричный метод (шаблон проектирования)

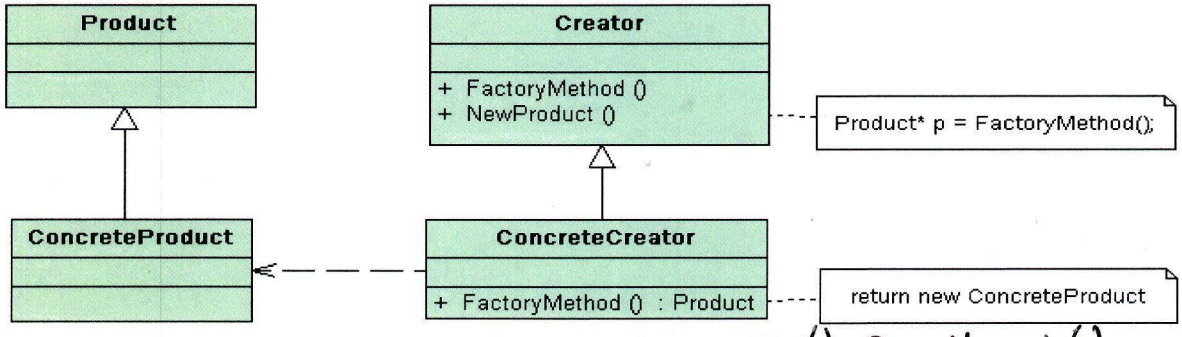
Фабричный метод (англ. *Factory Method* также известен как Виртуальный конструктор (англ. *Virtual Constructor*)) — порождающий шаблон проектирования, предоставляющий подклассам)(дочерним классам) интерфейс для создания экземпляров некоторого класса. Непосредственное решение о том, объект какого класса создавать происходит в подклассах. В момент создания наследники могут определить, какой класс создавать.
Делегирует) **создание** объектов **наследникам** родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Цель

Определяет интерфейс) для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя

Структура



p* → new() Product() class;

- ***Product*** — продукт
  - определяет интерфейс объектов создаваемых абстрактным методом;
- **ConcreteProduct** — конкретный продукт
  - реализует интерфейс Product;
- ***Creator*** — создатель
  - объявляет фабричный метод, который возвращает объект типа ***Product***. Может также содержать реализацию этого метода «по умолчанию»;
  - может вызывать фабричный метод для создания объекта типа ***Product***;
- **ConcreteCreator** — конкретный создатель
  - переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса **ConcreteProduct**.

Достоинства

- позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (**ConcreteProduct**), а оперируя лишь общим интерфейсом ***Product***;
- позволяет установить связь между параллельными иерархиями классов.

Недостатки

- необходимость создавать наследника ***Creator*** для каждого нового типа продукта **ConcreteProduct**.

```
// Абстрактный класс Creator определяет
abstract class Creator
{//абстрактный фабричный метод который возвращает объект Product.
public abstract Product FactoryMethod();
}

// Конкретные классы наследники для каждого конкретного класса продукта
// определяют свою реализацию метода FactoryMethod().
class ConcreteCreatorA : Creator
{
public override Product FactoryMethod()
{return new ConcreteProductA();
}}

// Причем метод возвращает определенный конкретный тип продукта.
class ConcreteCreatorB : Creator
{
public override Product FactoryMethod()
{return new ConcreteProductB();
}}

// Таким образом, класс Creator делегирует создание объекта Product своим наследникам.
// А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой
// конкретный тип продукта им создавать.
//////////////////////////////////////////
//Абстрактный класс Product определяет интерфейс класса, объекты которого надо создавать.
abstract class Product
{}
// Конкретные классы ConcreteProductA и ConcreteProductB представляют реализацию класса Product.
class ConcreteProductA : Product
{}
class ConcreteProductB : Product
{}
// Таких классов может быть множество
```

Теперь рассмотрим на реальном примере. Допустим, мы создаем программу для сферы строительства.

- Возможно, вначале мы захотим построить многоэтажный панельный дом. И для этого выбирается соответствующий подрядчик, который возводит каменные дома.
- Затем нам захочется построить деревянный дом и для этого также надо будет выбрать нужного подрядчика:

```
public static string EugeneMain()
{
AbstractFactory developer;
// "ООО КирпичСтрой"//////////////
developer = new ConcreteFactoryPanelDeveloper();
Product stone_house = developer.FactoryMethod();

//"Частный застройщик"/////////////
developer = new ConcreteFactoryWoodDeveloper();
Product wooden_house = developer.FactoryMethod();

return stone_house.Name + " + " + wooden_house.Name;
}

//////////////////////////////////////////
// абстрактный класс строительной компании Developer==Creator
abstract class AbstractFactory
{
////// фабричный метод ///
abstract public Product FactoryMethod();
}

//////////////////////////////////////////
// "ООО КирпичСтрой" == панельные дома
class ConcreteFactoryPanelDeveloper : AbstractFactory
{

public override Product FactoryMethod()
{return new PanelHouse();
}}
//////////////////////////////////////////
// "Частный застройщик" == деревянные дома
class ConcreteFactoryWoodDeveloper : AbstractFactory
{

public override Product FactoryMethod()
{return new WoodHouse();
}}

//////////////////////////////////////////
abstract class Product //House
{
public string Name{ get; set; }
}
//////////////////////////
class PanelHouse : Product //House
{
public PanelHouse()
{this.Name = "Панельный дом построен";
}}
//////////////////////////
class WoodHouse : Product //House
{
public WoodHouse()
{this.Name = "Деревянный дом построен";
}}
```

```
EugeneMain()
```

```
'Панельный дом построен + Деревянный дом построен'
```

В качестве абстрактного класса **Product** здесь выступает класс **Product-House**. Его две конкретные реализации

-
  - **PanelHouse** и
-
  - **WoodHouse**

представляют типы домов, которые будут строить подрядчики. В качестве абстрактного класса создателя выступает **AbstractFactory-Developer**, определяющий абстрактный метод **FactoryMethod()**. Этот метод реализуется в классах-наследниках

- **WoodDeveloper** и
- **PanelDeveloper**.

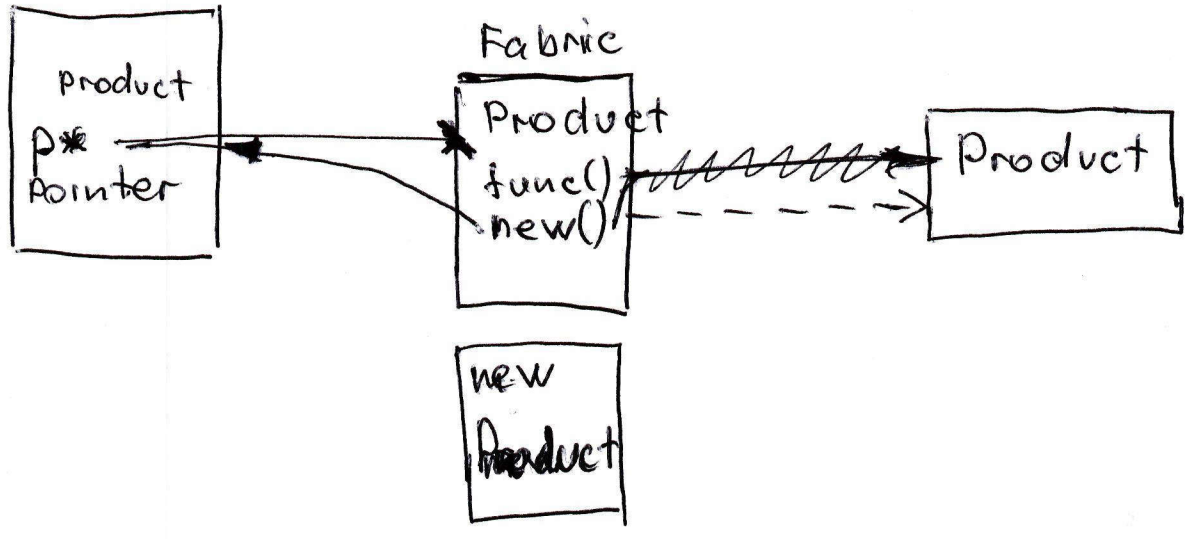
И если в будущем нам потребуется построить дома какого-то другого типа, например,

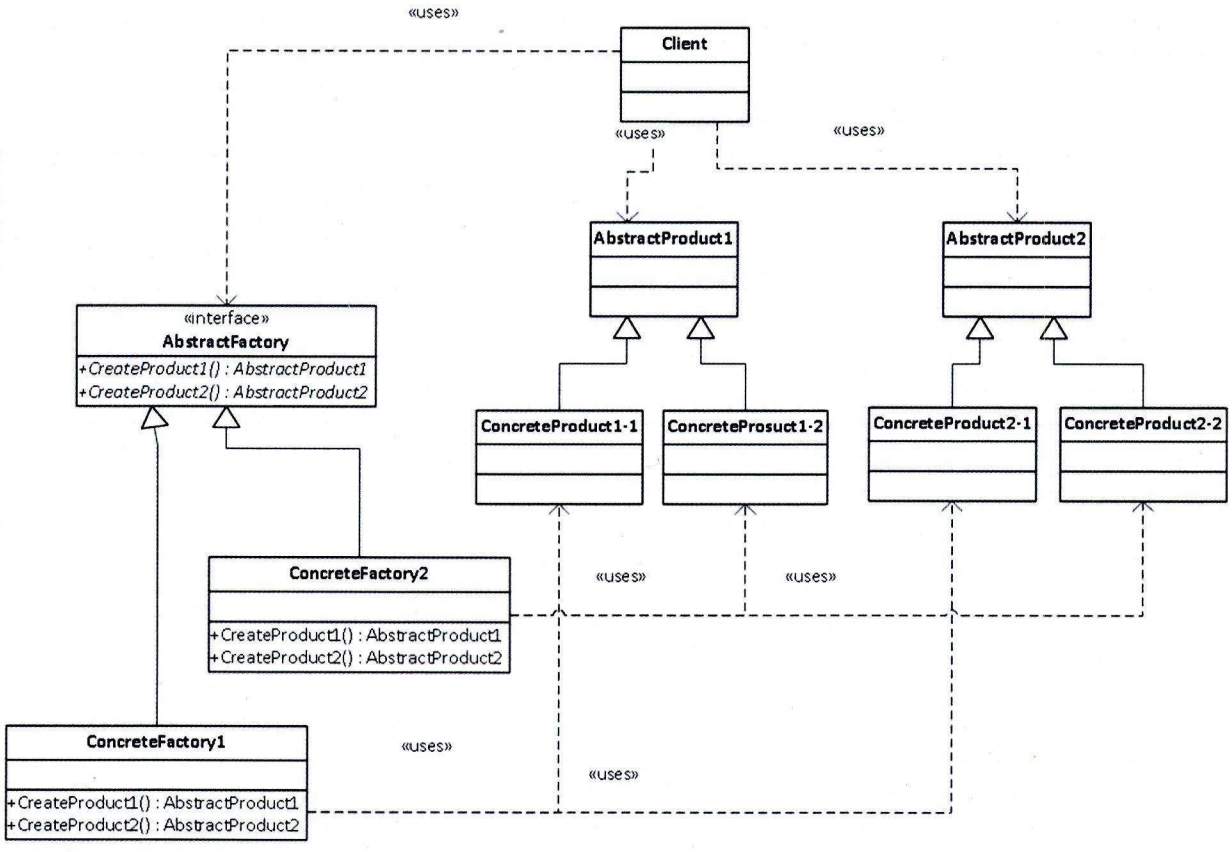
- кирпичные,

то мы можем с легкостью создать унаследованный от **Product** новый класс

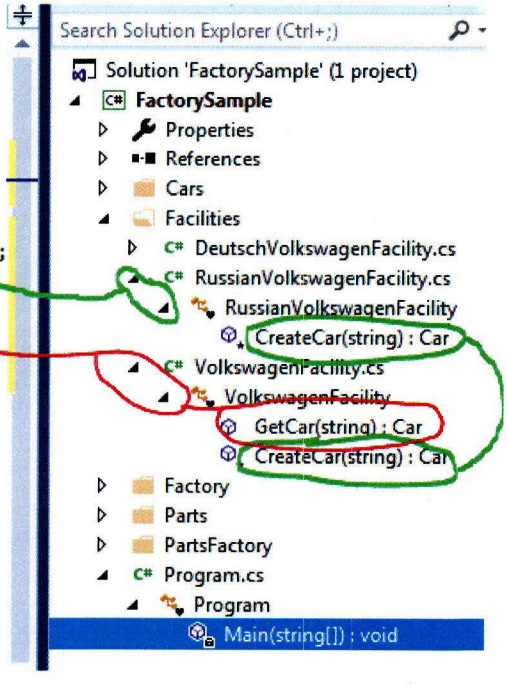
- **кирпичных-House** домов, и
- определить класс соответствующего **ConcreteFactory-подрядчика**.

Таким образом, система получится легко расширяемой. Правда, недостатки паттерна тоже очевидны - **для каждого нового продукта необходимо создавать свой класс создателя**.






```
using System;
using FactorySample.Facilities;
using FactorySample.Factory;

namespace FactorySample
{
class Program
{
static void Main(string[] args)
{VolkswagenFacility facility = new RussianVolkswagenFacility();

 facility.GetCar("Golf");
 Console.WriteLine();
 facility.GetCar("Passat");
 Console.WriteLine();
 facility.GetCar("Tiguan");
 Console.WriteLine();
 facility.GetCar("Touareg");
 Console.WriteLine();
 Console.ReadLine();
 }
 }}
```



p * → Factory
new()

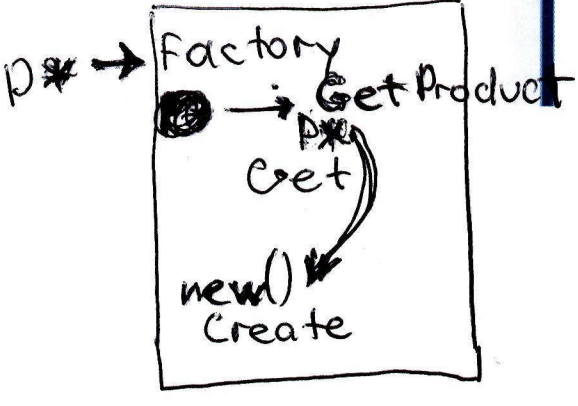
*Factory

```
using FactorySample.Cars;

namespace FactorySample.Facilities
{

abstract class VolkswagenFacility
{
public Car GetCar(string type)
{Car car = CreateCar(type);
 car.Configure();
 car.AssembleBody();
 car.InstallEngine();
 car.Paint();
 car.InstallWheels();
 return car;
}

protected abstract Car CreateCar(string type);
}}
```

- Facilities
  - DeutschVolkswagenFacility.cs
  - RussianVolkswagenFacility.cs
    - RussianVolkswagenFacility
      - CreateCar(string) : Car
  - VolkswagenFacility.cs
    - VolkswagenFacility
      - GetCar(string) : Car
      - CreateCar(string) : Car
- Factory
- Parts
- PartsFactory
- Program.cs
  - Program
    - Main(string[]) : void

```
using FactorySample.Cars;
using FactorySample.PartsFactory;

namespace FactorySample.Facilities
{

class RussianVolkswagenFacility : VolkswagenFacility
{
protected override Car CreateCar(string type)
{CarPartsFactory factory = new RussianCarPartsFactory();
 if(type == "Golf")
   return new Golf(factory);
 else if (type == "Passat")
   return new Passat(factory);
 else if (type == "Tiguan")
   return new Tiguan(factory);
 else if (type == "Touareg")
   return new Touareg(factory);

return null;
}
}}
```

- RussianVolkswagenFacility.cs
  - RussianVolkswagenFacility
    - CreateCar(string) : Car
- VolkswagenFacility.cs
  - VolkswagenFacility
    - GetCar(string) : Car
    - CreateCar(string) : Car
- Factory
- Parts
- PartsFactory
  - CarPartsFactory.cs
    - CarPartsFactory
  - DeutschCarPartsFactory.cs
  - RussianCarPartsFactory.cs
    - RussianCarPartsFactory
      - CreateEngine() : Engine
      - CreatePaint() : Paint
      - CreateWheels() : Wheels
- Program.cs

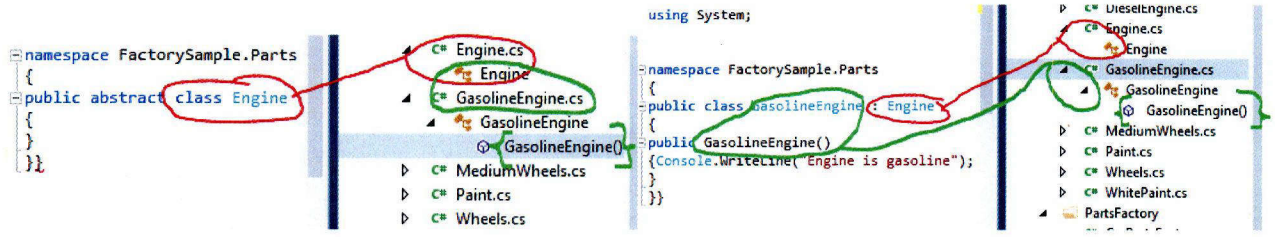


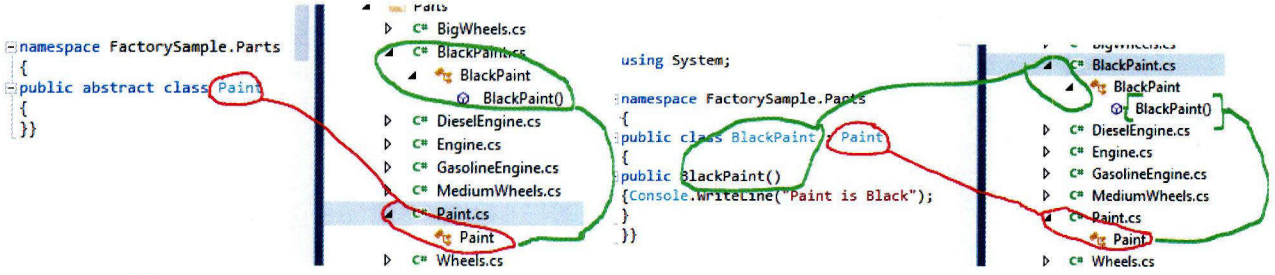
```
using FactorySample.Parts;

namespace FactorySample.PartsFactory
{

public abstract class CarPartsFactory
{
public abstract Engine CreateEngine();
public abstract Paint CreatePaint();
public abstract Wheels CreateWheels();
}}
```

- PartsFactory
  - CarPartsFactory.cs
    - CarPartsFactory
      - CreateEngine() : Engine
      - CreatePaint() : Paint
      - CreateWheels() : Wheels
  - DeutschCarPartsFactory.cs
  - RussianCarPartsFactory.cs
    - RussianCarPartsFactory
      - CreateEngine() : Engine
      - CreatePaint() : Paint
      - CreateWheels() : Wheels
  - Program.cs

```
using FactorySample.Parts;

namespace FactorySample.PartsFactory
{
public class RussianCarPartsFactory : Car
{
public override Engine CreateEngine()
{return new GasolineEngine();
}

public override Paint CreatePaint()
{return new BlackPaint();
}

public override Wheels CreateWheels()
{return new MediumWheels();
}
}}
```

- Parts
  - BigWheels.cs
  - BlackPaint.cs
  - DieselEngine.cs
  - Engine.cs
  - GasolineEngine.cs
  - MediumWheels.cs
  - Paint.cs
  - Wheels.cs
  - WhitePaint.cs
- PartsFactory
  - CarPartsFactory.cs
    - CarPartsFactory
      - CreateEngine() : Engine
      - CreatePaint() : Paint
      - CreateWheels() : Wheels
  - DeutschCarPartsFactory.cs
  - RussianCarPartsFactory.cs
    - RussianCarPartsFactory
      - CreateEngine() : Engine
      - CreatePaint() : Paint
      - CreateWheels() : Wheels





Configuring Golf
Body is: Hatchback
Engine is gasoline
Paint is Black
Wheels are 16"
Body is assembled
Engine is in its place
Car is painted
Wheels are installed

configure()
Assemble
Install
Paint
Install

Configuring Passat
Body is: Sedan
Engine is gasoline
Paint is Black
Wheels are 16"
Body is assembled
Engine is in its place
Car is painted
Wheels are installed

Configuring Tiguan
Body is: Crossover
Engine is gasoline
Paint is Black
Wheels are 16"
Body is assembled
Engine is in its place
Car is painted
Wheels are installed

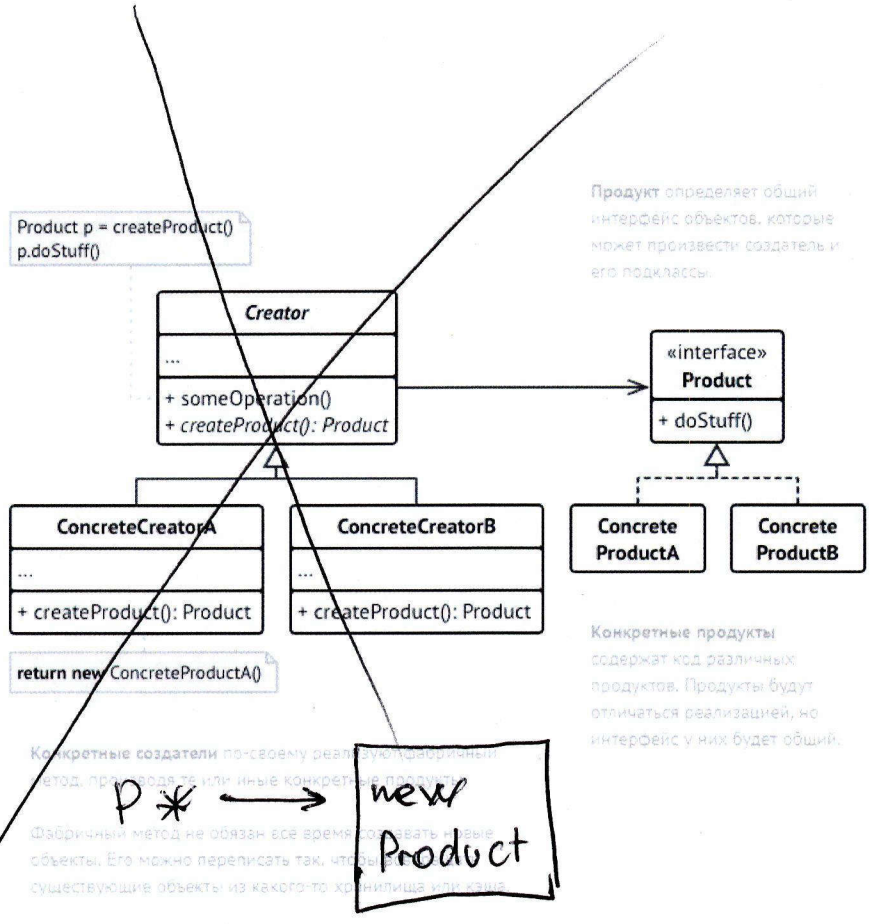
Configuring Touareg
Body is: Big Crossover
Engine is gasoline
Paint is Black
Wheels are 16"
Body is assembled
Engine is in its place
Car is painted
Wheels are installed

Структура

Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.



Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

Конкретные создатели по-своему реализуют фабричный метод, производя то или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы он возвращал существующие объекты из какого-то хранилища или кэша.

Псевдокод

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.

Концептуальный пример

Этот пример показывает структуру паттерна **Фабричный метод**, а именно — из каких классов он состоит, какие роли эти классы выполняют и как они взаимодействуют друг с другом.

Program.cs: Пример структуры паттерна

```
using System;

namespace RefactoringGuru.DesignPatterns.FactoryMethod.Conceptual
{

 // Класс Создатель объявляет фабричный метод, который должен возвращать
 // объект класса Продукт. Подклассы Создателя обычно предоставляют
 // реализацию этого метода.
 abstract class Creator
 {
 // Обратите внимание, что Создатель может также обеспечить реализацию
 // фабричного метода по умолчанию.
 public abstract IProduct FactoryMethod();

 // Также заметьте, что, несмотря на название, основная обязанность
 // Создателя не заключается в создании продуктов. Обычно он содержит
 // некоторую базовую бизнес-логику, которая основана  на объектах
 // Продуктов, возвращаемых фабричным методом.  Подклассы могут косвенно
 // изменять эту бизнес-логику, переопределяя фабричный метод и возвращая
 // из него другой тип продукта.
 public string SomeOperation()
 {// Вызываем фабричный метод, чтобы получить объект-продукт.
  var product = FactoryMethod();
  // Далее, работаем с этим продуктом.
  var result = "Creator: The same creator's code has just worked with "+ product.Operation();
  return result;
 }
 }

 // Конкретные Создатели переопределяют фабричный метод для того, чтобы
 // изменить тип результирующего продукта.
 class ConcreteCreator1 : Creator
 {// Обратите внимание, что сигнатура метода по-прежнему использует тип
  // абстрактного продукта, хотя  фактически из метода возвращается
  // конкретный продукт. Таким образом, Создатель может оставаться
  // независимым от конкретных классов продуктов.
  public override IProduct FactoryMethod()
  {return new ConcreteProduct1();
  }
 }

 class ConcreteCreator2 : Creator
 {
 public override IProduct FactoryMethod()
 {return new ConcreteProduct2();
 }
 }
```

new ()

new

```
// Интерфейс Продукта объявляет операции, которые должны выполнять все
// конкретные продукты.
public interface IProduct
{string Operation();
}

// Конкретные Продукты предоставляют различные реализации интерфейса
// Продукта.
class ConcreteProduct1 : IProduct
{
public string Operation()
{return "{Result of ConcreteProduct1}";
}
}

class ConcreteProduct2 : IProduct
{
public string Operation()
{return "{Result of ConcreteProduct2}";
}
}
```

```
class Client
{
public void Main()
{Console.WriteLine("App: Launched with the ConcreteCreator1.");
 ClientCode(new ConcreteCreator1());

 Console.WriteLine("");

 Console.WriteLine("App: Launched with the ConcreteCreator2.");
 ClientCode(new ConcreteCreator2());
}

// Клиентский код работает с экземпляром конкретного создателя, хотя и
// через его базовый интерфейс. Пока клиент продолжает работать с
// создателем через базовый интерфейс, вы можете передать ему любой
// подкласс создателя.
public void ClientCode(Creator creator)
{
 // ...
 Console.WriteLine("Client: I'm not aware of the creator's class," +
                   "but it still works.\n" + creator.SomeOperation());
 // ...
 }
}
```

```
class Program
{
static void Main(string[] args)
{
new Client().Main();
}
}
}
```

Output.txt: Результат выполнения

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of ConcreteProduct2}
```