**Фабричный метод**

# **Назначение и цель применения паттерна**

**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. В момент создания наследники могут определить, какой класс создавать. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Основное назначение: Для создания объектов различных типов одним интерфейсом.

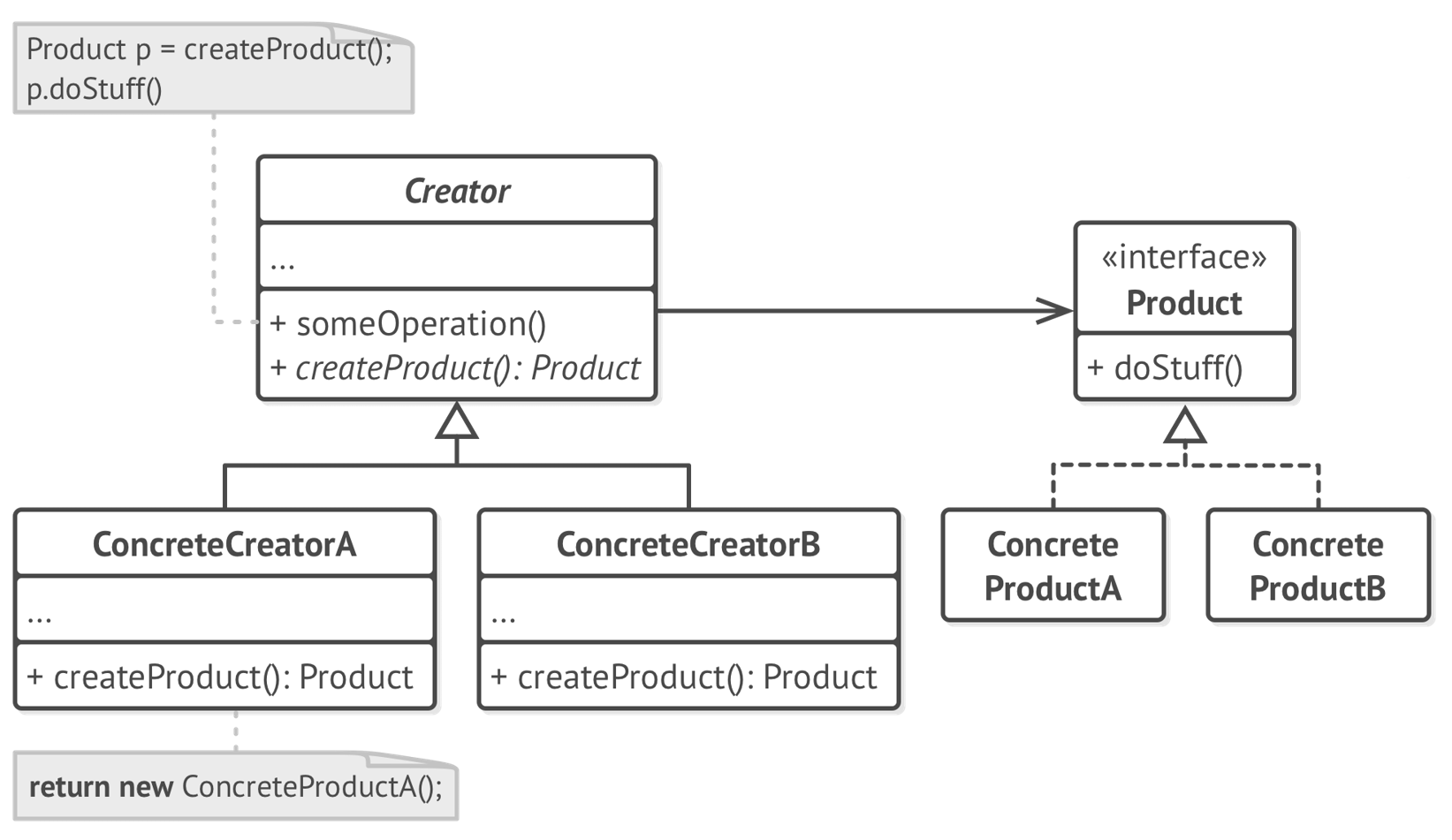
Цель применения паттерна: Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать создание подклассов.

Когда используется 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 диаграммы паттерна**

Для описания структуры паттерна Factory Method необходимо построить UML диаграмму классов.



UML диаграмма **Factory Method**

Основные элементы диаграммы:

* + - 1. **Интерфейс «Продукт»** определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
      2. **Конкретные продукты** содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
      3. **Создатель** объявляет фабричный метод, создающий объекты через общий интерфейс продуктов.
      4. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать фабричный метод по-своему.

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

# **Области применения**

В каких случаях можно применять паттерн Factory Method:

* + - 1. Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать код.
      2. Когда необходимо дать возможность пользователю расширять части фреймворка или библиотеки.
      3. Когда необходимо экономить системные ресурсы, повторно используя уже созданные объекты, вместо создания новых.

# **Особенности паттерна**

Достоинства паттерна:

* Избавляет класс от привязки к конкретным классам продуктов;
* Выделяет код производства продуктов в одно место, упрощая поддержку кода;
* Упрощает добавление новых продуктов в программу;
* Реализует принцип открытости/закрытости.

Недостатки паттерна:

* Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

# **Пример реализации**

Была разработана программа, автоматизирующая деятельность «Молочного завода».

abstract class DairyProduct

{

protected double fatness;

protected bool liquid;

public abstract string GetType();

public abstract string WayToMake();

public virtual double GetFatness()

{

return fatness;

}

public virtual void SetFatness(double f)

{

fatness = f;

}

public virtual void SetLiquid(bool l)

{

liquid = l;

}

public virtual string GetLiquid()

{

if (liquid)

return "Жидкий";

else

return "Твердый";

}

}

Создан абстрактный класс «Молочный продукт», для которого были определены два свойства: жирность и жидкость, а также методы для возвращения типа молочного продукта, способа производства молочного продукта, жирности продукта, жидкости, а также установка значений жирности и жидкости.

class MilkProduct : DairyProduct

{

public override string GetType()

{

return "Молоко";

}

public override string WayToMake()

{

return "Корову доят, продукт пастеризуется";

}

}

class CheaseProduct : DairyProduct

{

public override string GetType()

{

return "Сыр";

}

public override string WayToMake()

{

return "Плавления различных молочных продуктов";

}

}

class ButterProduct : DairyProduct

{

public override string GetType()

{

return "Масло";

}

public override string WayToMake()

{

return "Взбивание сметаны";

}

}

class YogurtProduct : DairyProduct

{

public override string GetType()

{

return "Йогурт";

}

public override string WayToMake()

{

return "Сбраживание кисломолочных продуктов";

}

}

От ранее созданного абстрактного класса «Молочный продукт» были созданы наследованные классы «Продукт Молоко», «Продукт Сыр», «Продукт Масло» и «Продукт Йогурт». Для каждого из них были переопределены методы возвращения типов и способов производства продукта.

abstract class DairyCreator

{

public abstract DairyProduct FactoryMethod();

}

Был создан абстрактный класс «Создатель молочной продукции», у которого был определен метод FactoryMethod. Данный метод абструктный, для каждого наследуемого класса метод будет переопределяться.

class MilkCreator : DairyCreator

{

public override DairyProduct FactoryMethod()

{

return new MilkProduct();

}

}

class CheaseCreator : DairyCreator

{

public override DairyProduct FactoryMethod()

{

return new CheaseProduct();

}

}

class ButterCreator : DairyCreator

{

public override DairyProduct FactoryMethod()

{

return new ButterProduct();

}

}

class YogurtCreator : DairyCreator

{

public override DairyProduct FactoryMethod()

{

return new YogurtProduct();

}

}

Для ранее каждого из созданных классов наследников «Молочный продукт» был опреден создатель, наследованный от класса «Создатель молочной продукции».

Был определен модуль для тестирования основных методов классов.

class Program

{

static void Main(string[] args)

{

DairyCreator[] creators = {

new MilkCreator(),

new CheaseCreator(),

new ButterCreator(),

new YogurtCreator()

};

foreach (DairyCreator cr in creators)

{

Console.WriteLine();

DairyProduct dP = cr.FactoryMethod();

Console.WriteLine("Создан: " + dP.GetType());

switch(dP.GetType())

{

case "Молоко":

dP.SetFatness(3.5);

dP.SetLiquid(true);

break;

case "Сыр":

dP.SetFatness(55.5);

dP.SetLiquid(false);

break;

case "Масло":

dP.SetFatness(75.5);

dP.SetLiquid(false);

break;

case "Йогурт":

dP.SetFatness(2.5);

dP.SetLiquid(true);

break;

}

Console.WriteLine("Жирность: " + dP.GetFatness());

Console.WriteLine("Состояние: " + dP.GetLiquid());

Console.WriteLine("Способ приготовления: " + dP.WayToMake());

}

Console.WriteLine();

Console.Read();

}