Фабричный метод

Фабричный метод определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом Фабричный метод делегирует операция создания экземпляра субклассом.

Класс-создатель не обладает информацией о фактическом типе создаваемых продуктов.
Задача Фабриного метода-перемешение создания экземплов в субклассы

Фабричный метод - паттерн, порождающий классы.
Назначение:

- Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Используйте паттерн фабричный метод, когда:

- Классу заранее неизвестно, объекты каких классов ему нужно создавать;
- Класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- Класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

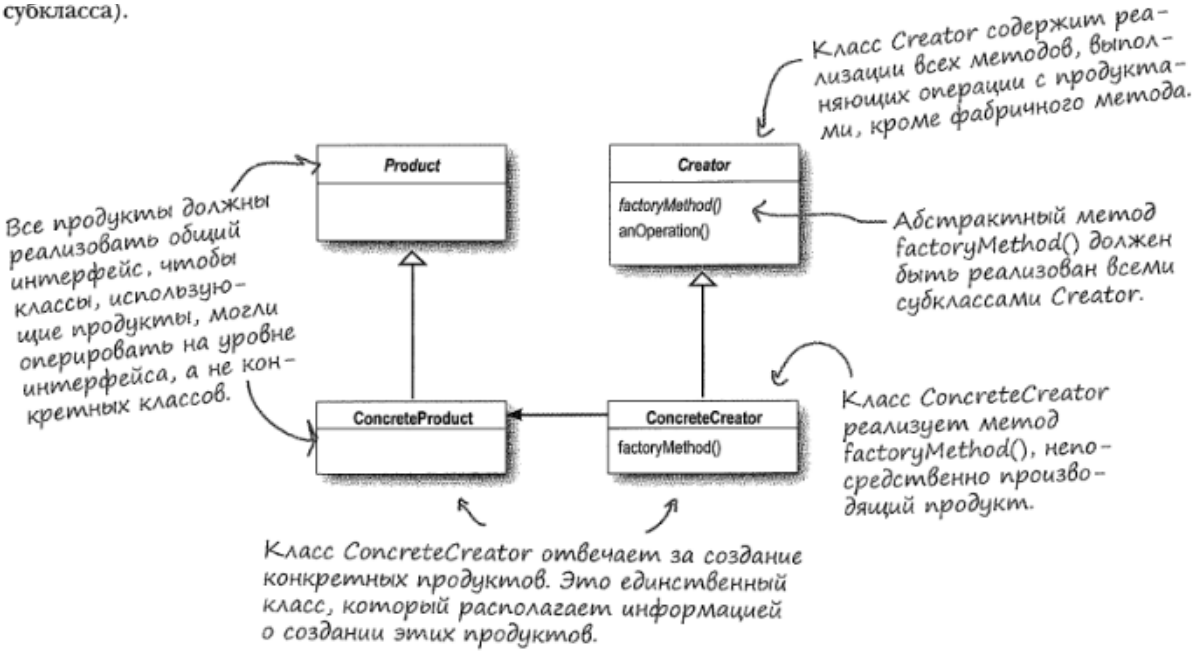
Достоинства:

- 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.

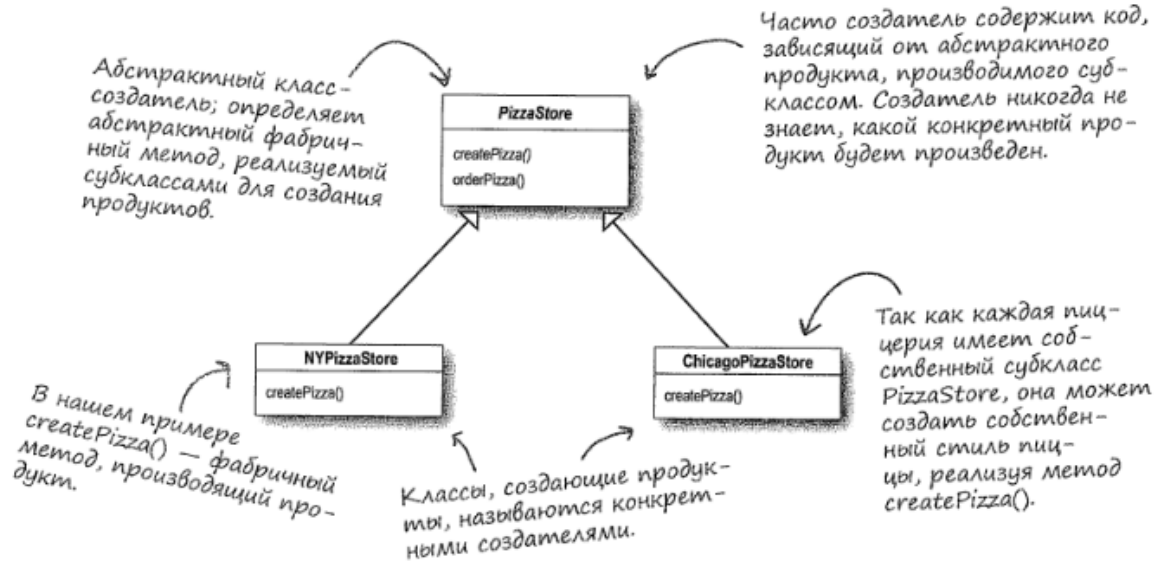
Недостатки:

- Необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct). Впрочем, современные языки программирования поддерживают конструкции, что позволяют реализовать фабричный метод без иерархии классов Creator.
-

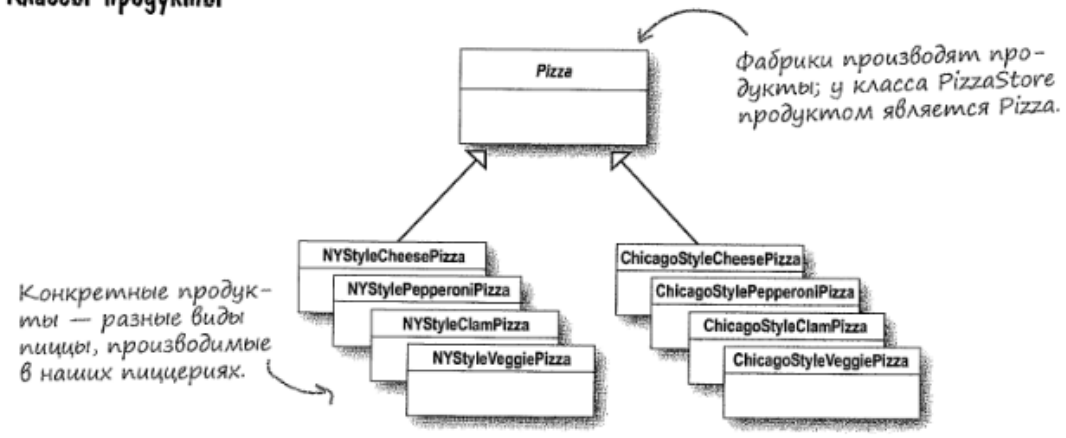
субкласса).



Классы-создатели



Классы-продукты



```cpp
#include <iostream>
#include <iostream>
#include <string.h>
using  namespace std;
class Pizza
{
protected:
    string name;
    string sause;
public:
    void prepare(){cout<<"preparing"<<endl;
    cout<<"adding "<<sause<<" on "<< name<<endl;};
    void cut(){};
    void bake(){};
    void box(){};
};
class NYCheese:public Pizza{
public:
    NYCheese()
    {
        name="NYCheese";
        sause="Marinara";
    }
};
class NYPeperony:public Pizza{
public:
    NYPeperony()
    {
        name="NYPeperony";
        sause="3333";
    }
};
class CHCheese:public Pizza{
public:
    CHCheese()
    {
        name="CHCheese";
        sause="Hines";
    }
};
class CHPeperony:public Pizza{
public:
    CHPeperony()
    {
        name="CHPeperony";
        sause="4444";
    }
};
class PizzaStore
{
public:
    Pizza* orderPizza(string type)
    {
        Pizza* pizza;
```

```
        pizza=createPizza(type);
        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
        return pizza;
    }
    virtual Pizza* createPizza(string type)=0;
};
class NyPizzaStore:public PizzaStore
{
public:
    Pizza* createPizza(string type)
    {
        if(type=="cheese")
            return new NYCheese();
        else
            return new NYPeperony();
    }
};
class CHPizzaStore:public PizzaStore
{
public:
    Pizza* createPizza(string type)
    {
        if(type=="cheese")
            return new CHCheese();
        else
            return new CHPeperony();
    }
};
int main() {
    PizzaStore* nystore=new NyPizzaStore();
    PizzaStore* chstore= new CHPizzaStore();
    Pizza* pz1=nystore->orderPizza("cheese");
    Pizza* pz2=chstore->orderPizza("xaxaxa");
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```