**КУРНОСЕНКО СОФЬЯ**

**№ 4 Применение порождающих паттернов проектирования**

**Задание**

1) Измените лабораторную №2. Создание и изменение объекта приложения сделайте на основе **Abstaract Factory и Builder.** Можно добавить функцию генерации *n* случайно заполненных объектов через запрос.

2) Создайте пользовательский тип, который хранит настройки приложения (цвет фона, шрифт, размер и т.п.) и сделайте его **Singleton**.

3) Добавьте функцию создания клона объекта на основе шаблона **Prototype**.

**Основы паттернов проектирования**

**Введение в паттерны проектирования**

Что представляют собой паттерны проектирования? **Паттерн** представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

Что же дает нам применение паттернов? При написании программ мы можем формализовать проблему в виде классов и объектов и связей между ними. И применить один из существующих паттернов для ее решения. В итоге нам не надо ничего придумывать. У нас уже есть готовый шаблон, и нам только надо его применить в конкретной программе.

Причем паттерны, как правило, не зависят от языка программирования. Их принципы применения будут аналогичны и в C#, и в Jave, и в других языках.

Также мышление паттернами упрощает групповую разработку программ. Зная применяемый паттерн проектирования и его основные принципы другому программисту будет проще понять его реализацию и использовать ее.

В то же время не стоит применять паттерны ради самих паттернов. Хорошая программа предполагает использование паттернов. Однако не всегда паттерны упрощают и улучшают программу. Неоправданное их использование может привести к усложнению программного кода, уменьшению его качества. Паттерн должен быть оправданным и эффективным способом решения проблемы.

Существует множество различных паттернов, которые решают разные проблемы и выполняют различные задачи. Но по своему действию их можно объединить в ряд групп. Рассмотрим некоторые группы паттернов. В основу классификации основных паттернов положена цель или задачи, которые определенный паттерн выполняет.

**Порождающие паттерны** — это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов. Среди них выделяются следующие:

* **Абстрактная фабрика (Abstract Factory)**
* **Строитель (Builder)**
* **Фабричный метод (Factory Method)**
* **Прототип (Prototype)**
* **Одиночка (Singleton)**

Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

Другая группа паттернов - **структурные паттерны** - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся:

* **Адаптер (Adapter)**
* **Мост (Bridge)**
* **Компоновщик (Composite)**
* **Декоратор (Decorator)**
* **Фасад (Facade)**
* **Приспособленец (Flyweight)**
* **Заместитель (Proxy)**

Третья группа паттернов называются **поведенческими** - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие:

* **Цепочка обязанностей (Chain of responsibility)**
* **Команда (Command)**
* **Интерпретатор (Interpreter)**
* **Итератор (Iterator)**
* **Посредник (Mediator)**
* **Хранитель (Memento)**
* **Наблюдатель (Observer)**
* **Состояние (State)**
* **Стратегия (Strategy)**
* **Шаблонный метод (Template method)**
* **Посетитель (Visitor)**

Существуют и другие классификации паттернов в зависимости от того, относится паттерн к классам или объектам.

Паттерны классов описывают отношения между классами посредством наследования. Отношения между классами определяются на стадии компиляции. К таким паттернам относятся:

* **Фабричный метод (Factory Method)**
* **Интерпретатор (Interpreter)**
* **Шаблонный метод (Template Method)**
* **Адаптер (Adapter)**

Другая часть паттернов - **паттерны объектов** описывают отношения между объектами. Эти отношения возникают на этапе выполнения, поэтому обладают большей гибкостью. К паттернам объектов относят следующие:

* **Абстрактная фабрика (Abstract Factory)**
* **Строитель (Builder)**
* **Прототип (Prototype)**
* **Одиночка (Singleton)**
* **Мост (Bridge)**
* **Компоновщик (Composite)**
* **Декоратор (Decorator)**
* **Фасад (Facade)**
* **Приспособленец (Flyweight)**
* **Заместитель (Proxy)**
* **Цепочка обязанностей (Chain of responsibility)**
* **Команда (Command)**
* **Итератор (Iterator)**
* **Посредник (Mediator)**
* **Хранитель (Memento)**
* **Наблюдатель (Observer)**
* **Состояние (State)**
* **Стратегия (Strategy)**
* **Посетитель (Visitor)**

И это только некоторые основные паттерны. А вообще различных шаблонов проектирования гораздо больше. Одни из них только начинают применяться, другие являются популярными на текущий момент, а некоторые уже менее распространены, чем раньше.

### Как выбрать нужный паттерн?

Прежде всего при решении какой-нибудь проблемы надо выделить все используемые сущности и связи между ними и абстрагировать их от конкретной ситуации. Затем надо посмотреть, вписывается ли абстрактная форма решения задачи в определенный паттерн. Например, суть решаемой задачи может состоять в создании новых объектов. В этом случае, возможно, стоит посмотреть на порождающие паттерны. Причем лучше не сразу взять какой-то определенный паттерн - первый, который показался нужным, а посмотреть на несколько родственных паттернов из одной группы, которые решают одну и ту же задачу.

При этом важно понимать смысл и назначение паттерна, явно представлять его абстрактную организацию и его возможные конкретные реализации. Один паттерн может иметь различные реализации, и чем чаще вы будете сталкиваться с этими реализациями, тем лучше вы будете понимать смысл паттерна. Но не стоит использовать паттерн, если вы его не понимаете, даже если он на первый взгляд поможет вам в решении задачи.

И в конечном счете надо придерживаться принципа KISS (Keep It Simple, Stupid) - сохранять код программы по возможности простым и ясным. Ведь смысл паттернов не в усложнении кода программы, а наоборот в его упрощении.

# Фабричный метод

**Также известен как:**Виртуальный конструктор, Factory Method



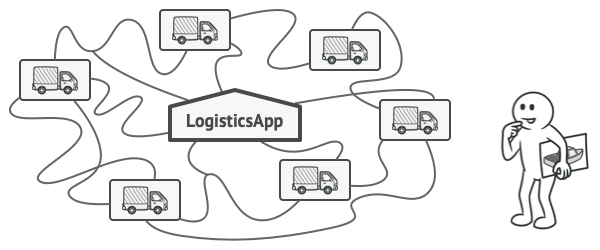
## Суть паттерна

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. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

## Проблема

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.



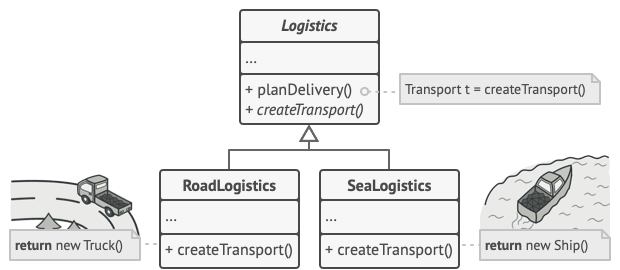
*Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.*

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

## Решение

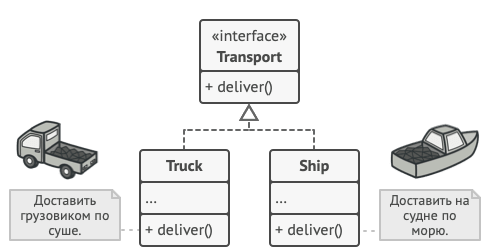
аттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого *фабричного* метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.



*Подклассы могут изменять класс создаваемых объектов.*

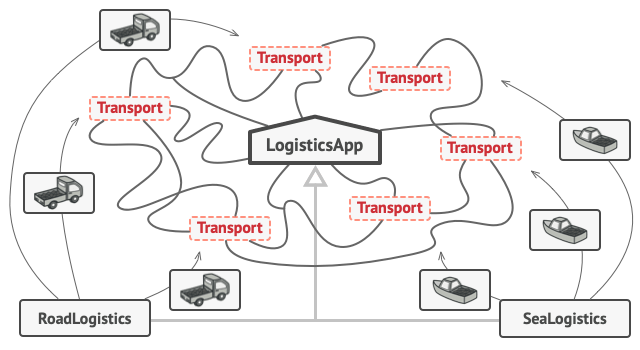
На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



*Все объекты-продукты должны иметь общий интерфейс.*

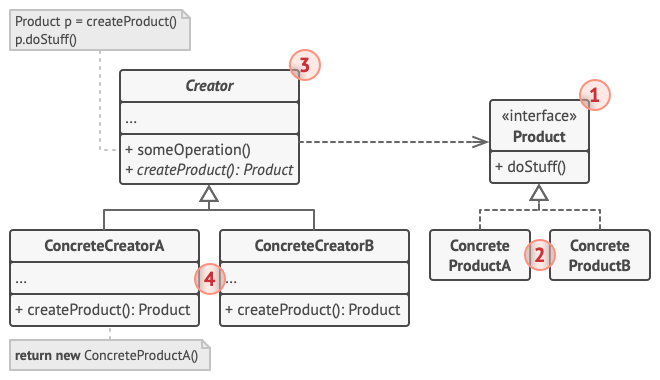
Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно.



*Пока все продукты реализуют общий интерфейс, их объекты можно взаимозаменять в клиентском коде.*

Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт. Для него будет важно, чтобы объект имел метод доставить, а как конкретно он работает — не важно.

## Структура



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

У всего множества конечных продуктов должен быть общий интерфейс, с которым и будет взаимодействовать пользователь.

Помимо этого общий интерфейс нужен, чтобы когда конкретные создатели возвращали новосозданные объекты, мы могли определить эти объекты, как объекты типа интерфейса. Соответственно и обращаться с ними, как с объектами одного типа.

1. **Конкретные продукты** содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
2. **Создатель (фабрика)** 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.

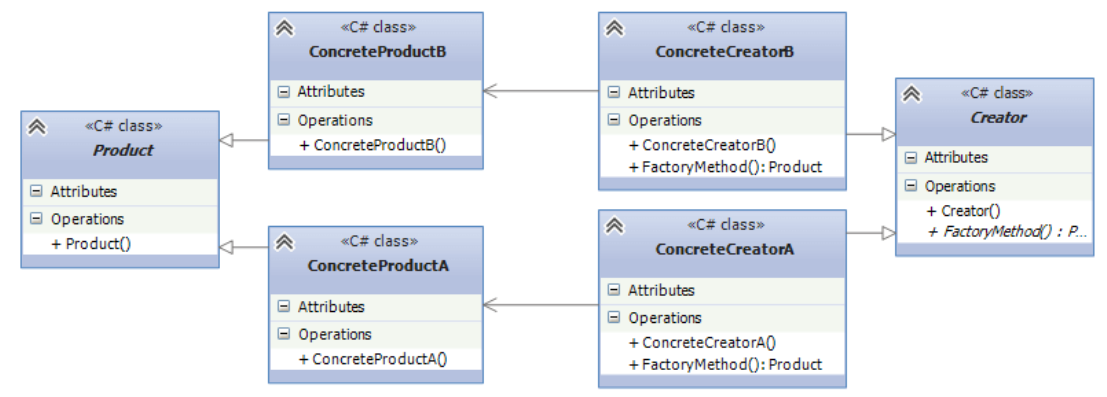
1. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

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

### 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

На языке UML паттерн можно описать следующим образом:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

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(); }

}

### Участники

* Абстрактный класс **Product** определяет интерфейс класса, объекты которого надо создавать.
* Конкретные классы **ConcreteProductA** и **ConcreteProductB** представляют реализацию класса Product. Таких классов может быть множество
* Абстрактный класс **Creator** определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
* Конкретные классы **ConcreteCreatorA** и **ConcreteCreatorB** - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.

## Концептуальный пример

Пример. Нам нужно создавать абонементы фитнес-центра в зависимости от пожеланий клиента. В данном коде мы можем создавать абонементы трёх типов: посещение зала, зал + бассейн, персональные тренировки.

using System;

// -----------------------

// интерфейс общий для всех производимых на фабрике продуктов, в нашем случае - для абонементов

internal interface IMembership

{

string Name { get; }

string Description { get; set; }

decimal GetPrice();

}

// класс абонемента на посещение зала

internal class GymMembership : IMembership

{

private readonly string \_name;

private readonly decimal \_price;

public GymMembership(decimal price)

{

\_name = "Gym membership";

\_price = price;

}

public string Name => \_name;

public string Description { get; set; }

public decimal GetPrice() => \_price;

}

// класс абонемента на посещение зала + бассейна

internal class GymPlusPoolMembership : IMembership

{

private readonly string \_name;

private readonly decimal \_price;

public GymPlusPoolMembership(decimal price)

{

\_name = "Gym + Pool membership";

\_price = price;

}

public string Name => \_name;

public string Description { get; set; }

public decimal GetPrice() => \_price;

}

// класс абонемента на посещение персональных тренировок

internal class PersonalTrainingMembership : IMembership

{

private readonly string \_name;

private readonly decimal \_price;

public PersonalTrainingMembership(decimal price)

{

\_name = "Personal Training membership";

\_price = price;

}

public string Name => \_name;

public string Description { get; set; }

public decimal GetPrice() => \_price;

}

// -----------------------

// Фабрика (родительский класс для дочерних фабрик)

internal abstract class MembershipFactory

{

// этот метод будет переопределяться в дочерних фабриках, то есть в тех, что отвечают за создание конкретных продуктов

// метод возвращает конкретный продукт, тип продукта IMembership, так как все продукты реализуют данный интерфейс

public abstract IMembership GetMembership();

}

// дочерняя фабрика для создания абонемента на посещение зала

internal class GymMembershipFactory : MembershipFactory

{

private readonly decimal \_price;

private readonly string \_description;

public GymMembershipFactory(decimal price, string description)

{

\_price = price;

\_description = description;

}

public override IMembership GetMembership()

{

GymMembership membership = new GymMembership(\_price)

{

Description = \_description

};

return membership;

}

}

// дочерняя фабрика для создания абонемента на посещение зала + бассейна

internal class GymPlusPoolMembershipFactory : MembershipFactory

{

private readonly decimal \_price;

private readonly string \_description;

public GymPlusPoolMembershipFactory(decimal price, string description)

{

\_price = price;

\_description = description;

}

public override IMembership GetMembership()

{

GymPlusPoolMembership membership = new GymPlusPoolMembership(\_price)

{

Description = \_description

};

return membership;

}

}

// дочерняя фабрика для создания абонемента на посещение персональных тренировок

internal class PersonalTrainingMembershipFactory : MembershipFactory

{

private readonly decimal \_price;

private readonly string \_description;

public PersonalTrainingMembershipFactory(decimal price, string description)

{

\_price = price;

\_description = description;

}

public override IMembership GetMembership()

{

PersonalTrainingMembership membership = new PersonalTrainingMembership(\_price)

{

Description = \_description

};

return membership;

}

}

// -----------------------

// использование созданных классов

class Program

{

static void Main()

{

// приветствуем пользователя и просим его ввести тип абонемента, который требуется создать

Console.WriteLine(">>> Welcome to FitnessClub CRM <<<\n");

Console.WriteLine("> Enter the membership type you would like to create: ");

Console.WriteLine("> G - Gym");

Console.WriteLine("> P - Gym + Pool");

Console.WriteLine("> T - Personal Training");

string membershipType = Console.ReadLine();

// возвращенный методом объект записываем в переменную типа MembershipFactory

MembershipFactory factory = GetFactory(membershipType);

if(factory != null) // проверка на случай, если пользователь ввел неверный символ и объект дочерней фабрики не создался

{

// вызовом метода GetMembership у дочерней фабрики создаем объект конкретного абонемента

IMembership membership = factory.GetMembership();

// выводим информацию о созданном абонементе

Console.WriteLine("\n> Membership you've just created: \n");

Console.WriteLine(

$"\tName:\t\t{membership.Name}\n" +

$"\tDescription:\t{membership.Description}\n" +

$"\tPrice:\t\t{membership.GetPrice()}");

Console.ReadLine();

}

}

// данный метод, в зависимости от ввода пользователя, создает объект необходимой дочерней фабрики

// в этой фабрике будут храниться данные, переданные в конструктор

// далее при вызове у этого объекта дочерней фабрики метода GetMembership, данные запишутся в объект

// конкретного продукта

private static MembershipFactory GetFactory(string membershipType) =>

membershipType.ToLower() switch // ToLower() - переводим введенный символ в нижний регистр, чтобы не зависеть от регистра выбранного пользователем

{

"g" => new GymMembershipFactory(100, "Basic membership"),

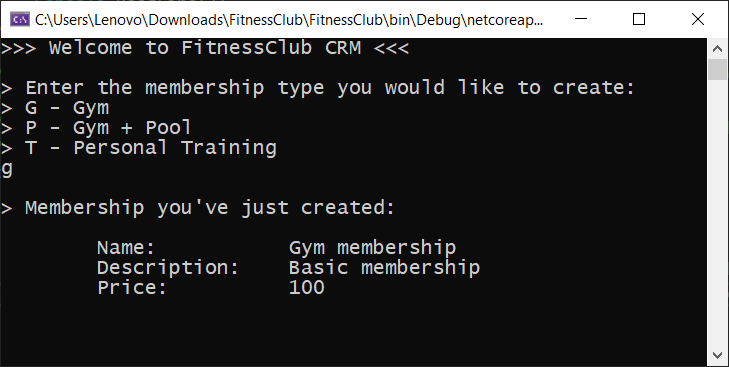
"p" => new GymPlusPoolMembershipFactory(250, "Good price membership"),

"t" => new PersonalTrainingMembershipFactory(400, "Best for professionals"),

\_ => null // если введено что-то другое, кроме предложенных символов

};

}

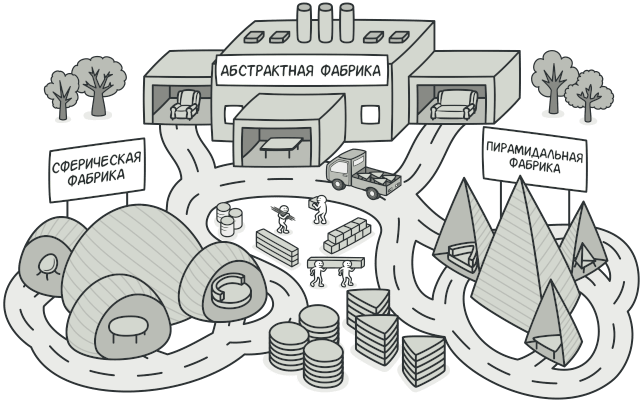


<https://www.youtube.com/watch?v=fudZFG-Cm0Y&list=LL&index=2>

# Абстрактная фабрика

**Также известен как:**Abstract Factory

**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



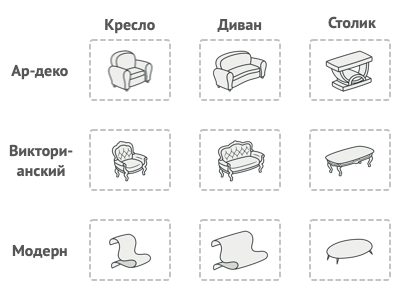
### Когда использовать абстрактную фабрику

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

## Проблема

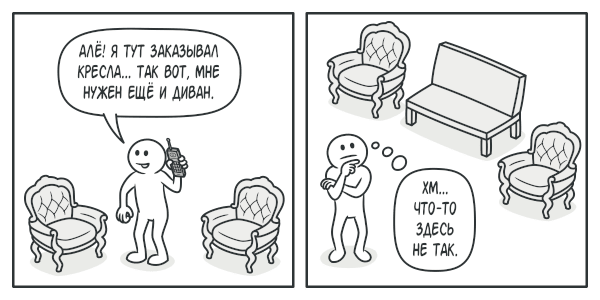
Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.



*Семейства продуктов и их вариации.*

Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

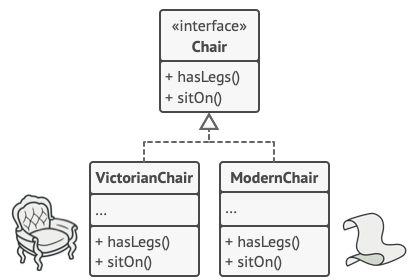


*Клиенты расстраиваются, если получают несочетающиеся продукты.*

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейcтв в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

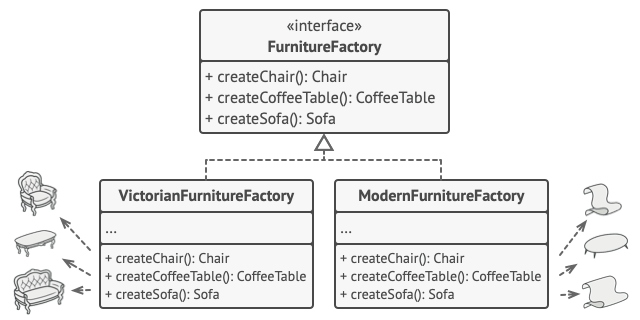
## Решение

Для начала паттерн Абстрактная фабрика предлагает **выделить общие интерфейсы для отдельных продуктов**, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.



*Все вариации одного и того же объекта должны жить в одной иерархии классов.*

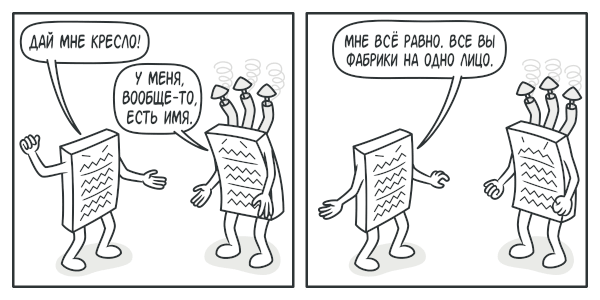
**Далее вы создаёте *абстрактную фабрику*** — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.



*Конкретные фабрики соответствуют определённой вариации семейства продуктов.*

Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн,ДиваныМодерн и СтоликиМодерн.

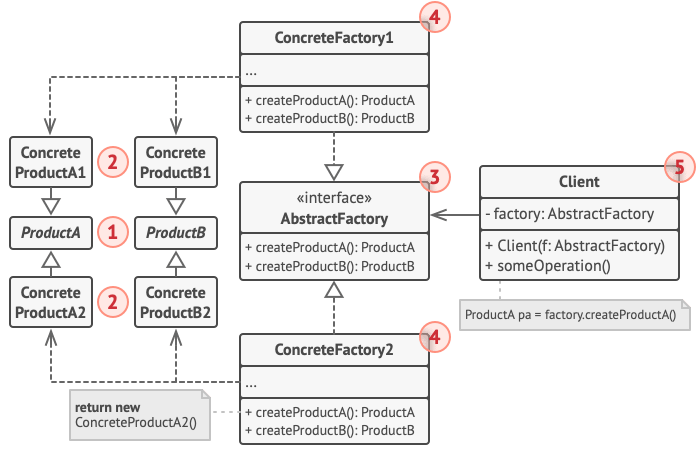
Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.



*Для клиентского кода должно быть безразлично, с какой фабрикой работать.*

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

## Структура



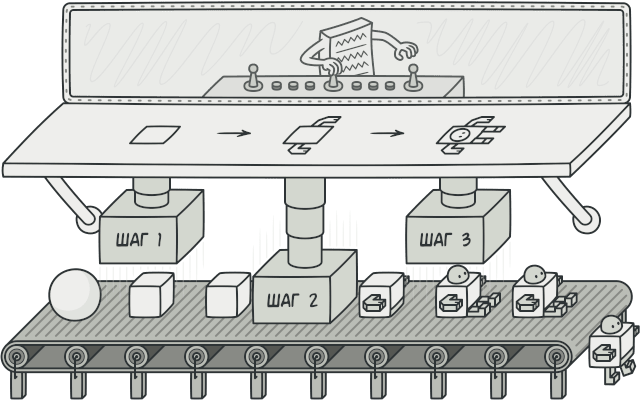
1. **Абстрактные продукты** объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
2. **Конкретные продукты** — большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).
3. **Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).
4. **Конкретные фабрики** относятся каждая к своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

**ABSTRACT FACTORY vs FACTORY METHOD**

# Строитель

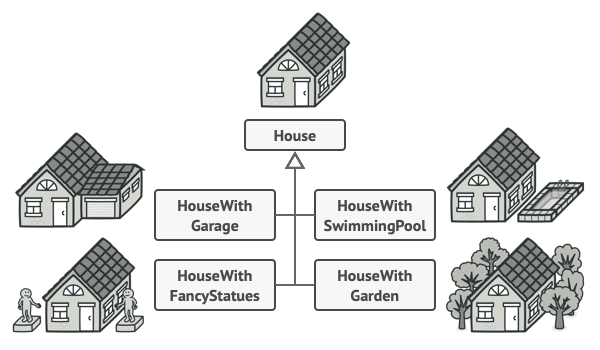
**Также известен как:**Builder

**Строитель** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



## Проблема

Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже — распылён по всему клиентскому коду.

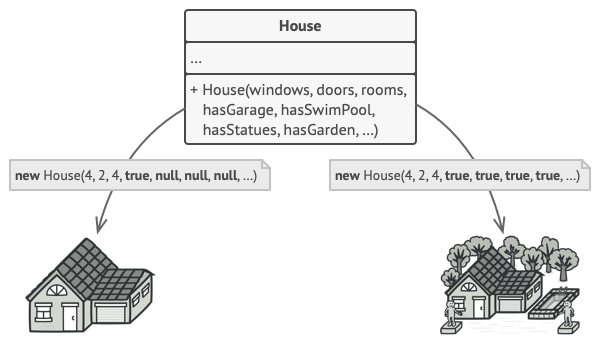


*Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.*

Например, давайте подумаем о том, как создать объект Дом. Чтобы построить стандартный дом, нужно поставить 4 стены, установить двери, вставить пару окон и положить крышу. Но что, если вы хотите дом побольше да посветлее, имеющий сад, бассейн и прочее добро?

Самое простое решение — расширить класс Дом, создав подклассы для всех комбинаций параметров дома. Проблема такого подхода — это громадное количество классов, которые вам придётся создать. Каждый новый параметр, вроде цвета обоев или материала кровли, заставит вас создавать всё больше и больше классов для перечисления всех возможных вариантов.

Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор Дома, принимающий уйму параметров для контроля над создаваемым продуктом. Действительно, это избавит вас от подклассов, но приведёт к другой проблеме.

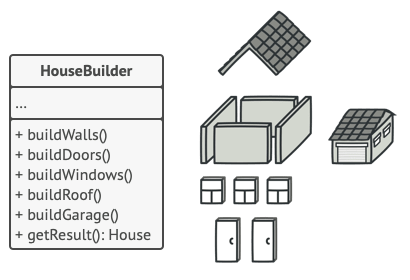


*Конструктор со множеством параметров имеет свой недостаток: не все параметры нужны большую часть времени.*

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за [**длинного списка параметров**](https://refactoring.guru/ru/smells/long-parameter-list). К примеру, далеко не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно в 99% случаев.

## Решение

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.

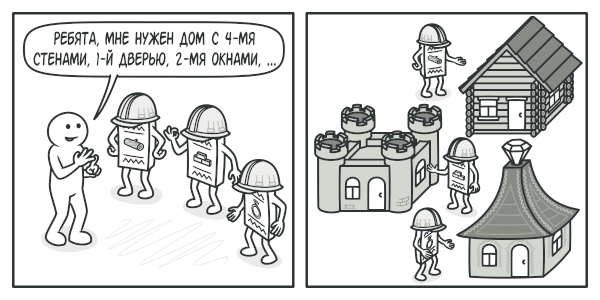


*Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.*

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построитьСтены, вставитьДвери и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. Например, деревянный дом потребует строительства стен из дерева, а каменный — из камня.

В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.

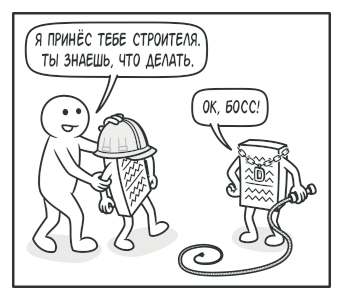


*Разные строители выполнят одну и ту же задачу по-разному.*

Например, один строитель делает стены из дерева и стекла, другой из камня и железа, третий из золота и бриллиантов. Вызвав одни и те же шаги строительства, в первом случае вы получите обычный жилой дом, во втором — маленькую крепость, а в третьем — роскошное жилище. Замечу, что код, который вызывает шаги строительства, должен работать со строителями через общий интерфейс, чтобы их можно было свободно взаимозаменять.

#### Директор

Вы можете пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их.

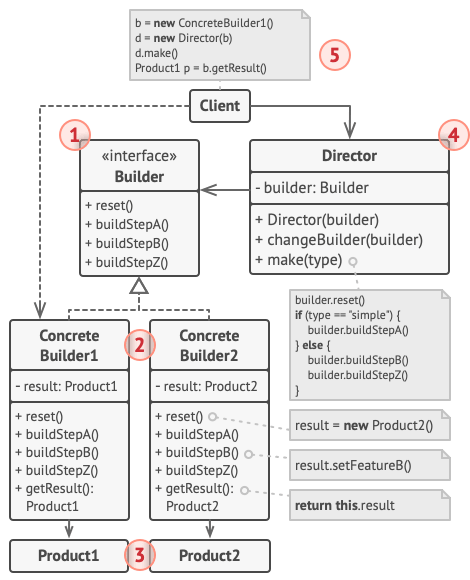


*Директор знает, какие шаги должен выполнить объект-строитель, чтобы произвести продукт.*

Отдельный класс директора не является строго обязательным. Вы можете вызывать методы строителя и напрямую из клиентского кода. Тем не менее, директор полезен, если у вас есть несколько способов конструирования продуктов, отличающихся порядком и наличием шагов конструирования. В этом случае вы сможете объединить всю эту логику в одном классе.

Такая структура классов полностью скроет от клиентского кода процесс конструирования объектов. Клиенту останется только привязать желаемого строителя к директору, а затем получить у строителя готовый результат.

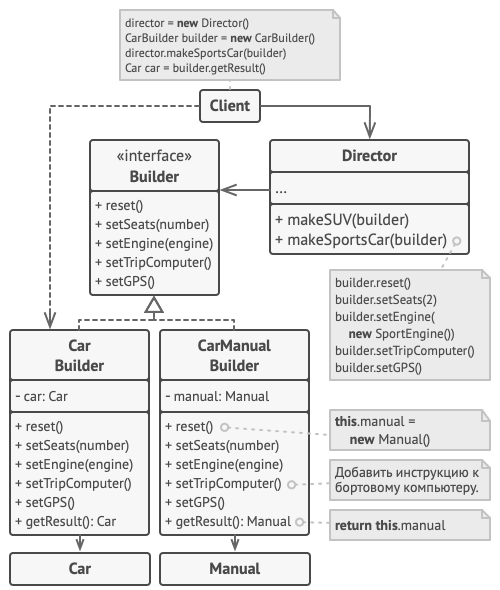
**Структура**



1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. **Продукт** — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
5. Обычно **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

## Псевдокод

В этом примере **Строитель** используется для пошагового конструирования автомобилей, а также технических руководств к ним.



*Пример пошагового конструирования автомобилей и инструкций к ним.*

Автомобиль — это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того чтобы настраивать автомобиль через конструктор, мы вынесем его сборку в отдельный класс-строитель, предусмотрев методы для конфигурации всех частей автомобиля.

Клиент может собирать автомобили, работая со строителем напрямую. Но, с другой стороны, он может поручить это дело директору. Это объект, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций автомобилей.

Но к каждому автомобилю нужно ещё и руководство, совпадающее с его конфигурацией. Для этого мы создадим ещё один класс строителя, который вместо конструирования автомобиля, будет печатать страницы руководства к той детали, которую мы встраиваем в продукт. Теперь, пропустив оба типа строителей через одни и те же шаги, мы получим автомобиль и подходящее к нему руководство пользователя.

Очевидно, что бумажное руководство и железный автомобиль — это две разных вещи, не имеющих ничего общего. По этой причине мы должны получать результат напрямую от строителей, а не от директора. Иначе нам пришлось бы жёстко привязать директора к конкретным классам автомобилей и руководств.

## Применимость

**→ Когда вы хотите избавиться от «телескопического конструктора».**

 Допустим, у вас есть один конструктор с десятью опциональными параметрами. Его неудобно вызывать, поэтому вы создали ещё десять конструкторов с меньшим количеством параметров. Всё, что они делают — это переадресуют вызов к базовому конструктору, подавая какие-то значения по умолчанию в параметры, которые пропущены в них самих.

**class** **Pizza** {

Pizza(**int** size) { ... }

Pizza(**int** size, **boolean** cheese) { ... }

Pizza(**int** size, **boolean** cheese, **boolean** pepperoni) { ... }

// ...

*Такого монстра можно создать только в языках, имеющих механизм перегрузки методов, например, C# или Java.*

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны. А значит, больше не нужно пытаться «запихнуть» в конструктор все возможные опции продукта.

**→ Когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.**

 Строитель можно применить, если создание нескольких представлений объекта состоит из одинаковых этапов, которые отличаются в деталях.

Интерфейс строителей определит все возможные этапы конструирования. Каждому представлению будет соответствовать собственный класс-строитель. А порядок этапов строительства будет задавать класс-директор.

## Шаги реализации

1. Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
2. Опишите эти шаги в общем интерфейсе строителей.
3. Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства.

Не забудьте про метод получения результата. Обычно конкретные строители определяют собственные методы получения результата строительства. Вы не можете описать эти методы в интерфейсе строителей, поскольку продукты не обязательно должны иметь общий базовый класс или интерфейс. Но вы всегда сможете добавить метод получения результата в общий интерфейс, если ваши строители производят однородные продукты с общим предком.

1. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
2. Клиентский код должен будет создавать и объекты строителей, и объект директора. Перед началом строительства клиент должен связать определённого строителя с директором. Это можно сделать либо через конструктор, либо через сеттер, либо подав строителя напрямую в строительный метод директора.
3. Результат строительства можно вернуть из директора, но только если метод возврата продукта удалось поместить в общий интерфейс строителей. Иначе вы жёстко привяжете директора к конкретным классам строителей.

**Преимущества и недостатки**

**Преимущества:**

* Позволяет создавать продукты пошагово.
* Позволяет использовать один и тот же код для создания различных продуктов.
* Изолирует сложный код сборки продукта от его основной бизнес-логики.

**Недостатки:**

* Усложняет код программы из-за введения дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

## Концептуальный пример

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