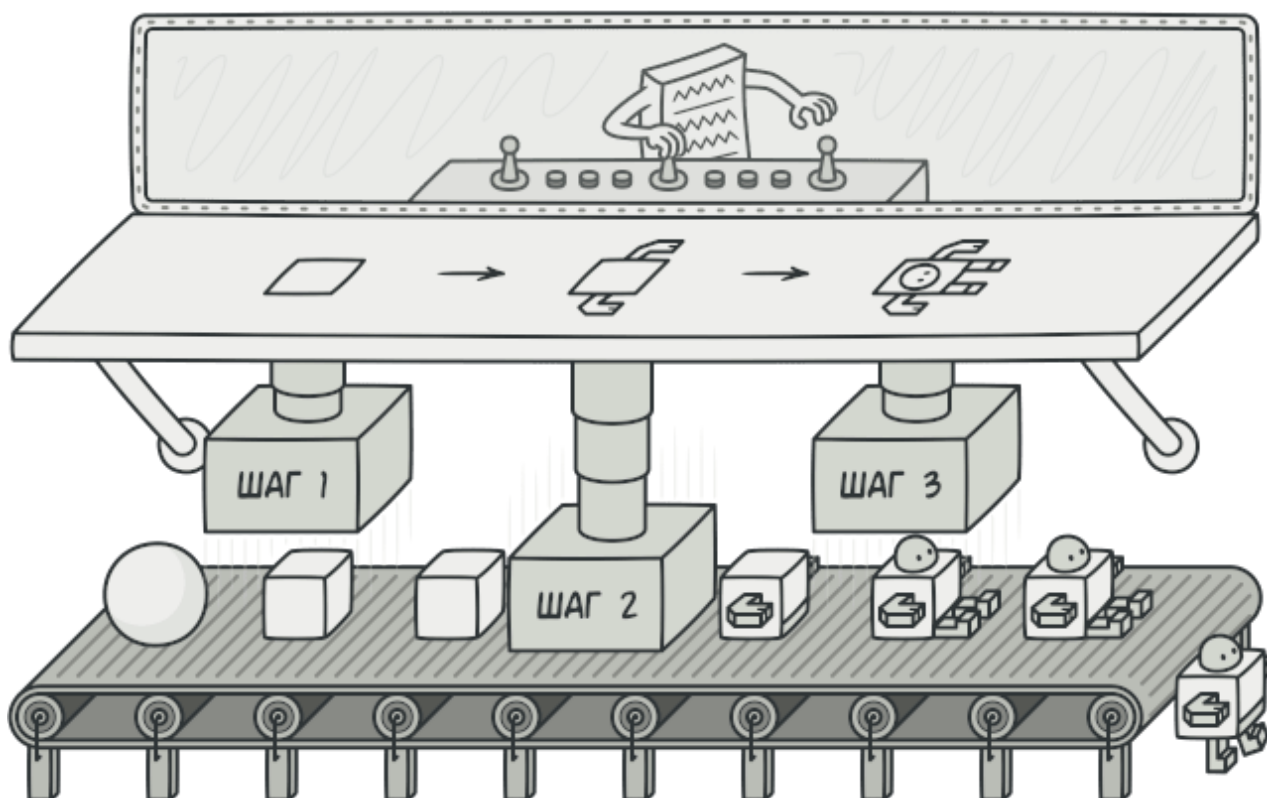


Строитель

Также известен как: Builder

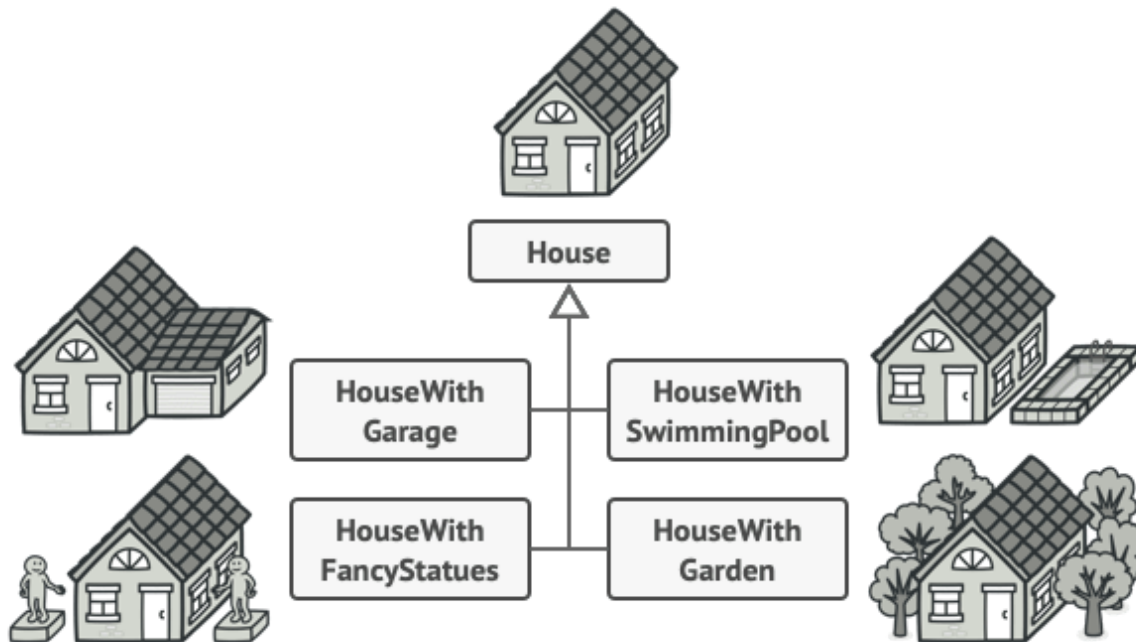
Суть паттерна

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.

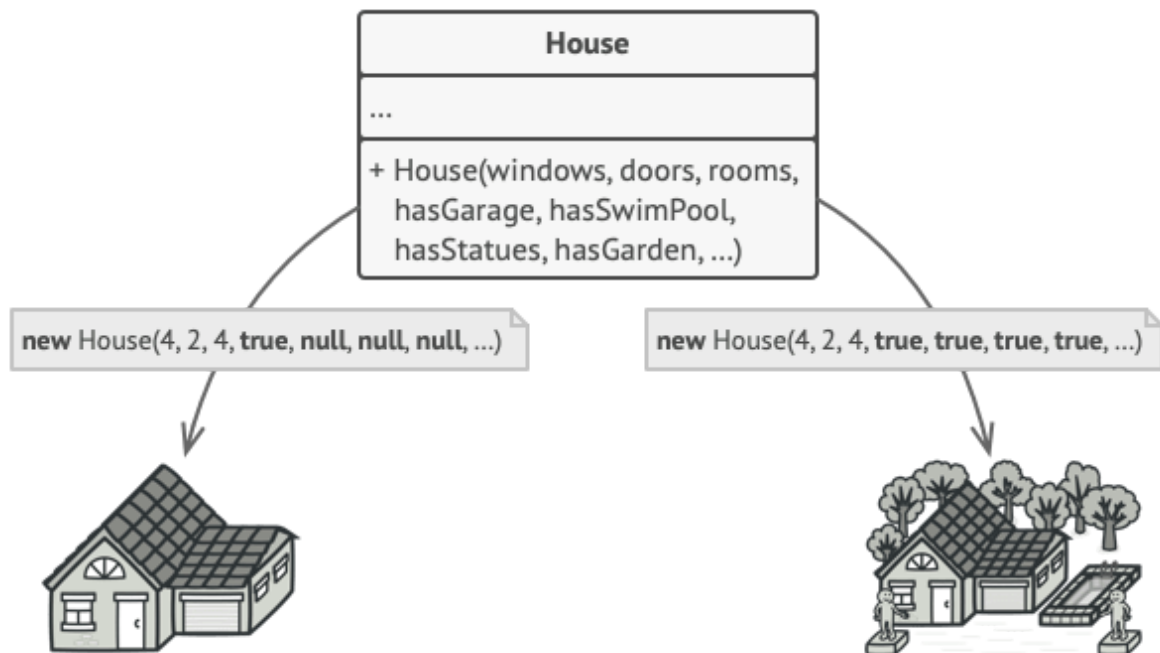


Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.

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.

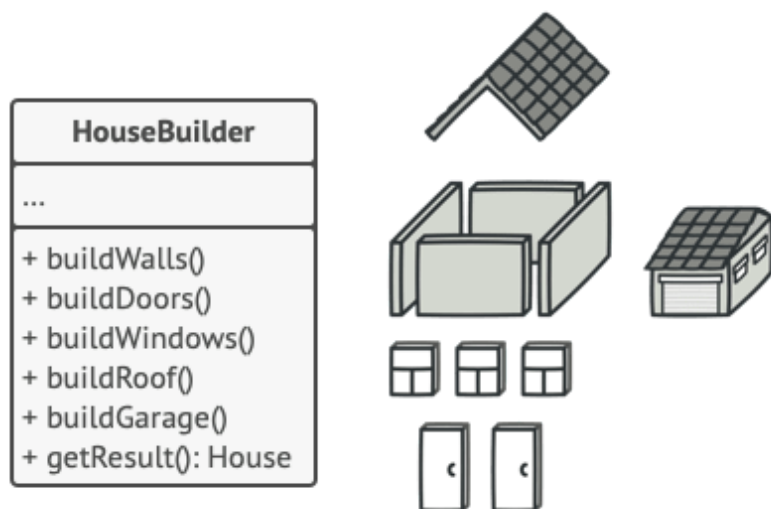


Конструктор со множеством параметров имеет свой недостаток: не все параметры нужны большую часть времени.

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за длинного списка параметров. 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*.

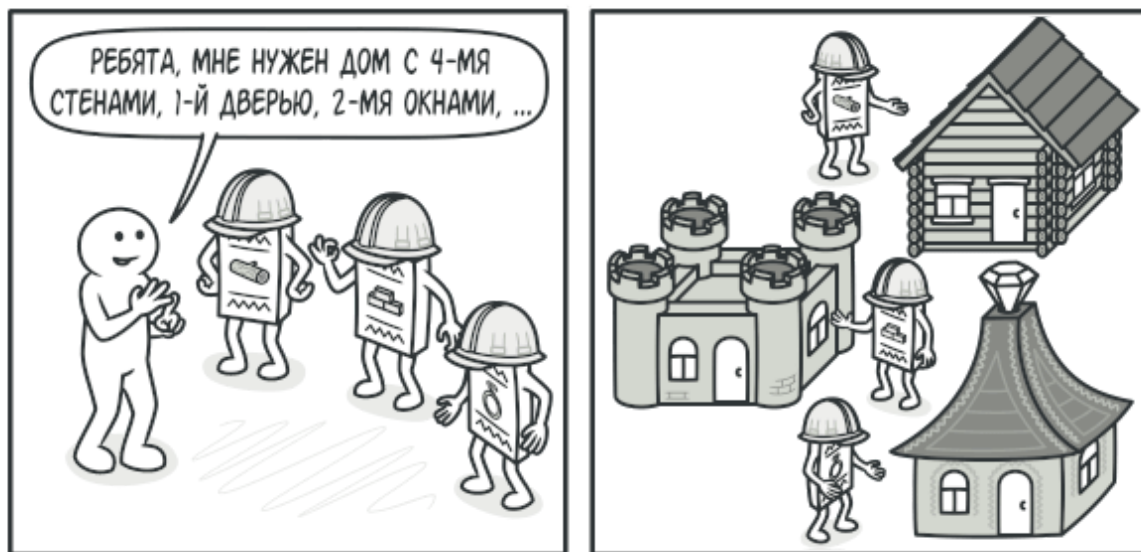


Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построить Стены, вставить Двери 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 выполняют 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.

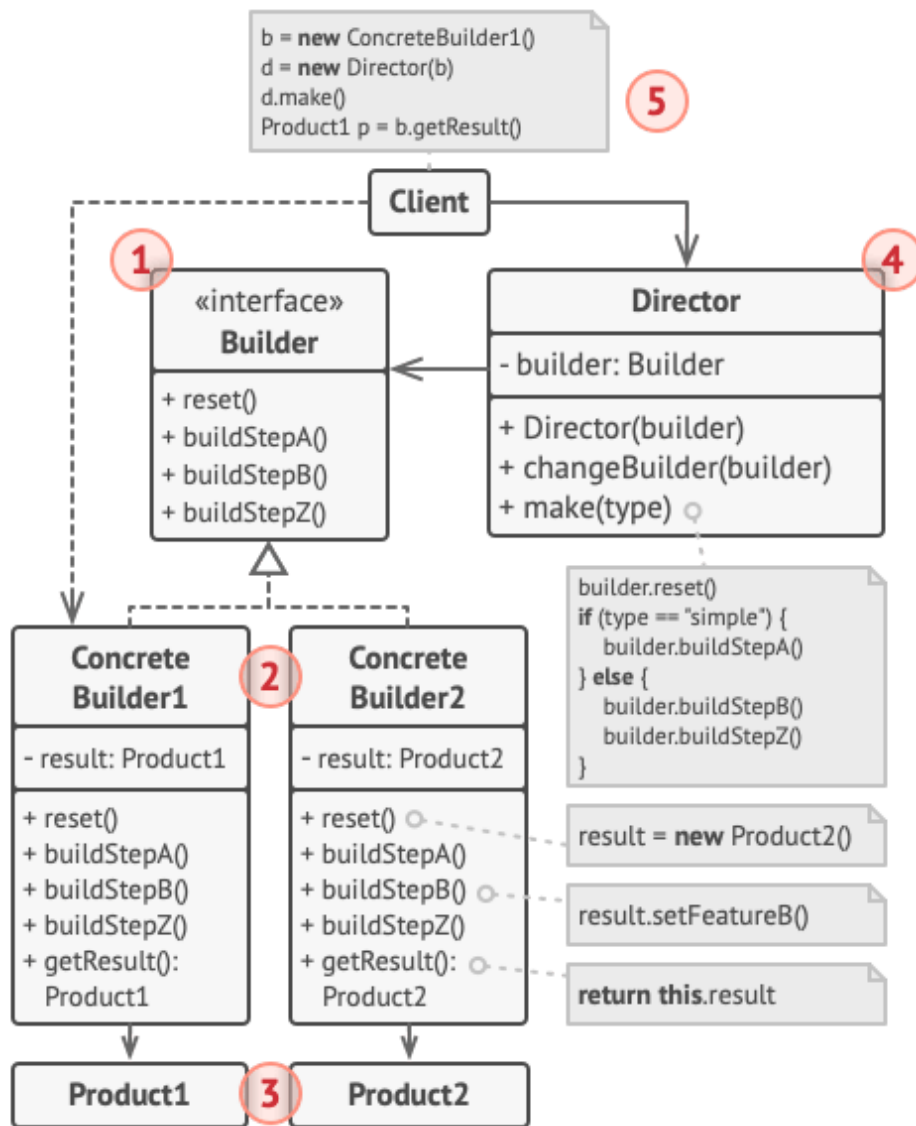


Директор знает, какие шаги должен выполнить объект-строитель, чтобы произвести продукт.

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.

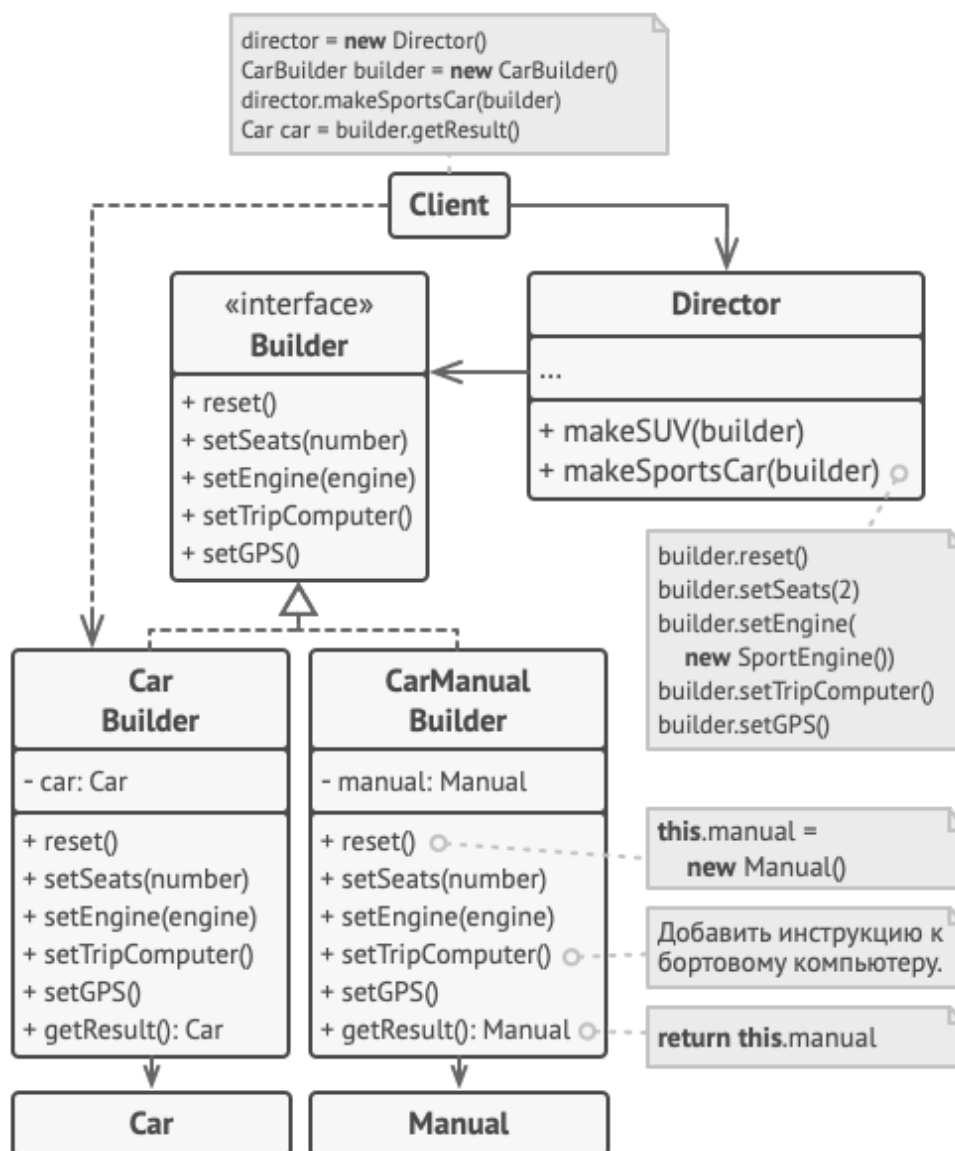
Структура



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.

```
// Строитель может создавать различные продукты, используя один
// и тот же процесс строительства.
```

```
class Car is
```

```
    // Автомобили могут отличаться комплектацией: типом
    // двигателя, количеством сидений, могут иметь или не иметь
    // GPS и систему навигации и т. д. Кроме того, автомобили
    // могут быть городскими, спортивными или внедорожниками.
```

```
class Manual is
```

```
    // Руководство пользователя для данной конфигурации
    // автомобиля.
```

```
// Интерфейс строителя объявляет все возможные этапы и шаги
// конфигурации продукта.
```

```
interface Builder is
```

```
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)
```

```
// Все конкретные строители реализуют общий интерфейс по-своему.
```

```
class CarBuilder implements Builder is
```

```
    private field car:Car
    method reset()
        // Поместить новый объект Car в поле "car".
    method setSeats(...) is
        // Установить указанное количество сидений.
    method setEngine(...) is
        // Установить поданный двигатель.
    method setTripComputer(...) is
        // Установить поданную систему навигации.
    method setGPS(...) is
        // Установить или снять GPS.
    method getResult():Car is
        // Вернуть текущий объект автомобиля.
```

```
// В отличие от других порождающих паттернов, где продукты
// должны быть частью одной иерархии классов или следовать
// общему интерфейсу, строители могут создавать совершенно
// разные продукты, которые не имеют общего предка.
```

```
class CarManualBuilder implements Builder is
```

```
    private field manual:Manual
    method reset()
        // Поместить новый объект Manual в поле "manual".
    method setSeats(...) is
```



```

    // Описать, сколько мест в машине.
method setEngine(...) is
    // Добавить в руководство описание двигателя.
method setTripComputer(...) is
    // Добавить в руководство описание системы навигации.
method setGPS(...) is
    // Добавить в инструкцию инструкцию GPS.
method getResult():Manual is
    // Вернуть текущий объект руководства.

// Директор знает, в какой последовательности нужно заставлять
// работать строителя, чтобы получить ту или иную версию
// продукта. Заметьте, что директор работает со строителем через
// общий интерфейс, благодаря чему он не знает тип продукта,
// который изготавливает строитель.
class Director is
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

// Директор получает объект конкретного строителя от клиента
// (приложения). Приложение само знает, какого строителя нужно
// использовать, чтобы получить определённый продукт.
class Application is
    method makeCar() is
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getResult()

        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)

        // Готовый продукт возвращает строитель, так как
        // директор чаще всего не знает и не зависит от
        // конкретных классов строителей и продуктов.
        Manual manual = builder.getResult()

```

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.

Когда вам нужно собирать сложные составные объекты, например, деревья Компоновщика.

Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого не построить древовидную структуру, вроде Компоновщика.

Заметьте, что Строитель не позволяет посторонним объектам иметь доступ к конструируемому объекту, пока тот не будет полностью готов. Это предохраняет клиентский код от получения незаконченных «битых» объектов.

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.

4. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
5. 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.
6. 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

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.

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.

Отношения с другими паттернами

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).
- Строитель концентрируется на построении сложных объектов шаг за шагом. Абстрактная фабрика специализируется на создании семейств связанных продуктов. *Строитель* возвращает продукт только после выполнения всех шагов, а *Абстрактная фабрика* возвращает продукт сразу же.
- Строитель позволяет пошагово сооружать дерево Компоновщика.
- Паттерн Строитель может быть построен в виде Моста: *директор* будет играть роль абстракции, а *строители* — реализации.

- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.