Строитель

Builder

Суть паттерна

Строитель:

- Порождающий
- Создаёт сложные объекты пошагово
- Даёт возможность использовать один и тот же код для получения разных представлений объектов.

Зачем ?

Представлю объект, требующий долгой инициализации.

Это объект класса, в конструкторе которого очень много параметров. Мало того, что при вызове много параметров, так ещё и инициализация всего этого занимает много места внутри самого конструктора.

Проверяйте:

```java
public class House {
    private int windowsCount;
    private int doorsCount;
    private int roomsCount;
    private boolean hasGarage;
    private boolean hasPool;
    private int floorsCount;

    public House(int w, int d, int r, boolean g, boolean p, int f) {
        this.doorsCount = d;
        this.windowsCount = w;
        this.hasGarage = g;
        this.hasPool = p;
        this.floorsCount = f;
        this.roomsCount = r;
    }

    @Override
    public String toString() {
        return this.doorsCount + "\n" + this.windowsCount + "...";
    }
}
```

```java
var house = new House( w: 10,  d: 3,  r: 7,  g: true,  p: false,  f: 1);
```

Зачем ?

- Часть этих параметров - **простаивает**,
- Вызовы конструктора - **выглядят «плохо»** из-за длинного списка параметров.

Не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно (имея значения false) в подавляющем большинстве случаев.

Решение

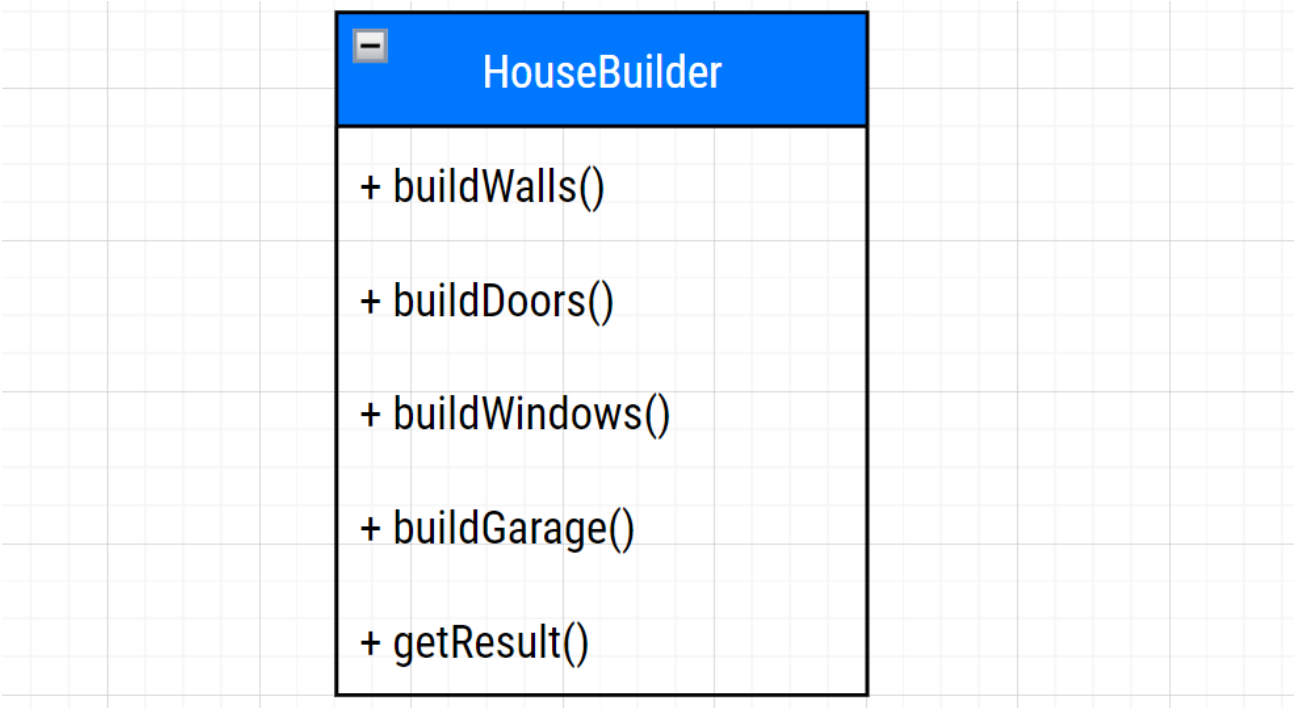
Разбиваем процесс конструирования объекта на отдельные шаги (например, «построить стены», «вставить двери» и другие).

По очереди вызываем нужные методы. Причём вызывать все необязательно.

Только те, что нужны.

Решение

Всё, теперь поручаю процесс строительства отдельным объектам за пределами моего класса !



Решение

Незатронутым полям можно по умолчанию присваивать какие-то дефолтные значения.

Таким образом вызываю только те компоненты, которые хочу подкорректировать/определить

```
// теперь можно использовать строителей
var builder = new HouseBuilder();
var builtWithBuilder :House = builder
                                .setFloorsCount(5)
                                .setWindowsCount(10)
                                .setDoorsCount(3)
                                .setRoomsCount(7)
                                .getHouse();
```

Решение

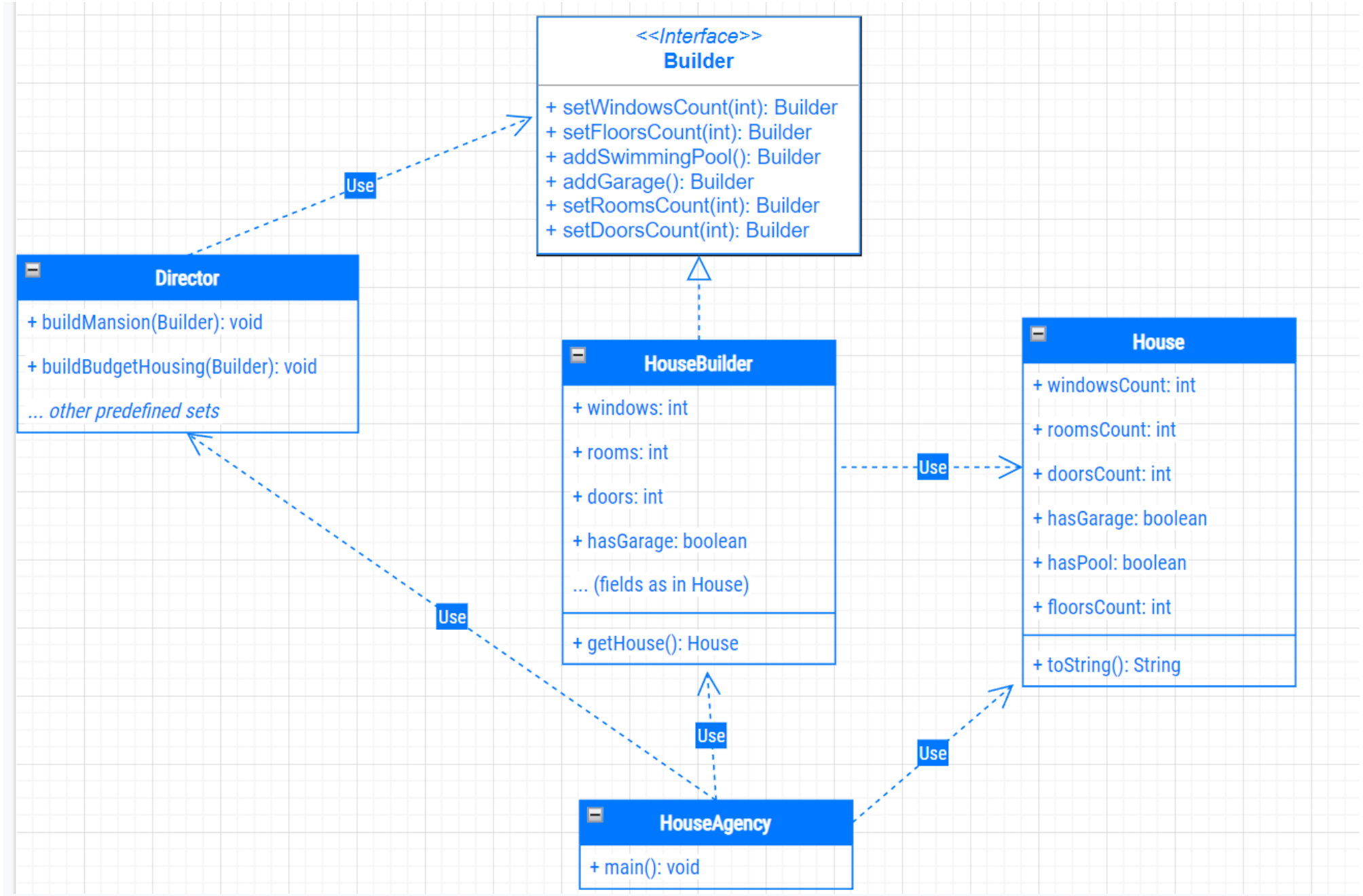
«Директора» позволят быстро создавать какие-то категории конфигураций.

```
// или можно использовать "директора"
// в нём есть готовые конфигурации для строительства
var director = new Director();
director.buildBudgetHousing(builder);
System.out.println(builder.getHouse());
```

Решение

Или так. «Фундамент» взял из готового сета и дополнительно добавил к нему какую-то доп. опцию

```
// или можно использовать "директора"
// в нём есть готовые конфигурации для строительства
var director = new Director();
director.buildBudgetHousing(builder);
builder.addGarage();
System.out.println(builder.getHouse());
```


<<Interface>>
Builder
+ setWindowsCount(int): Builder
+ setFloorsCount(int): Builder
+ addSwimmingPool(): Builder
+ addGarage(): Builder
+ setRoomsCount(int): Builder
+ setDoorsCount(int): Builder
Use
Director
+ buildMansion(Builder): void
+ buildBudgetHousing(Builder): void
... other predefined sets
HouseBuilder
+ windows: int
+ rooms: int
+ doors: int
+ hasGarage: boolean
... (fields as in House)
+ getHouse(): House
Use
House
+ windowsCount: int
+ roomsCount: int
+ doorsCount: int
+ hasGarage: boolean
+ hasPool: boolean
+ floorsCount: int
+ toString(): String
Use
Use
Use
HouseAgency
+ main(): void

Решение

Live coding или обзор существующего кода ?

Что-то ещё ?

⚖ 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.