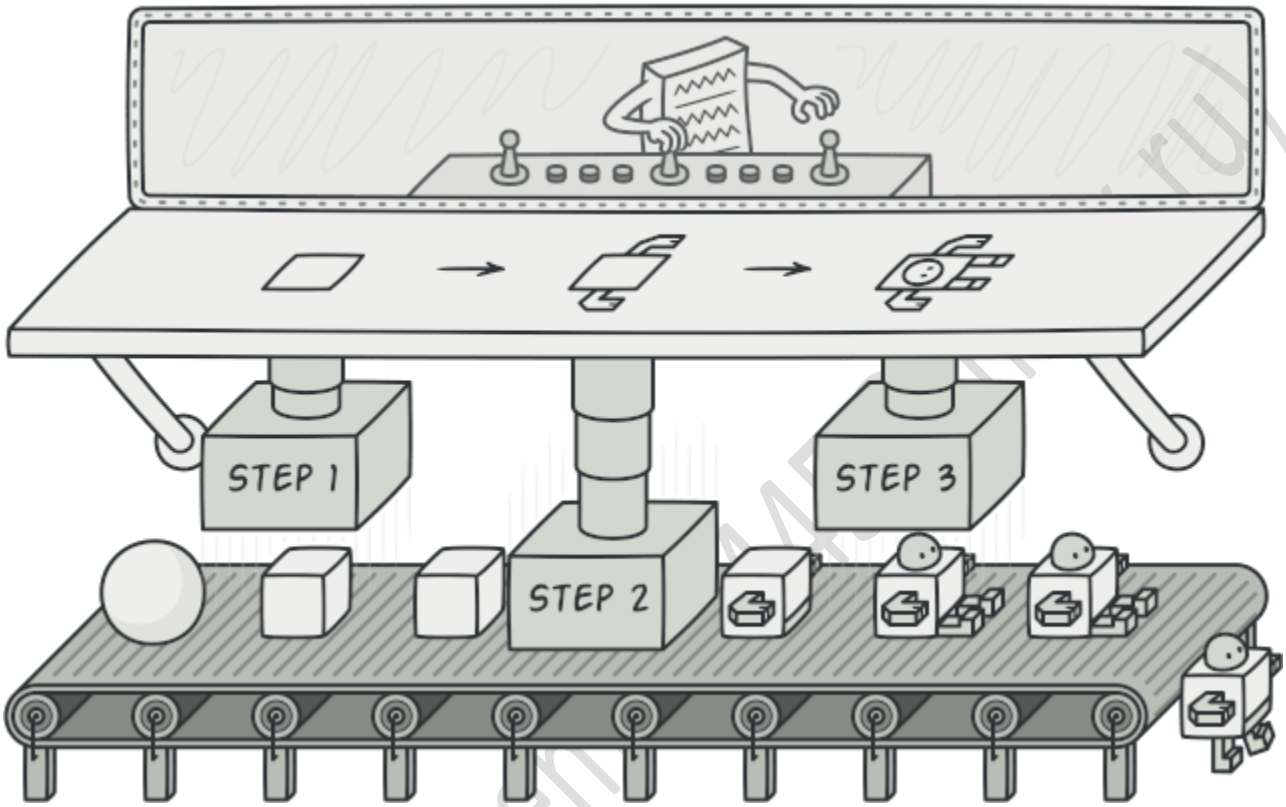
Builder

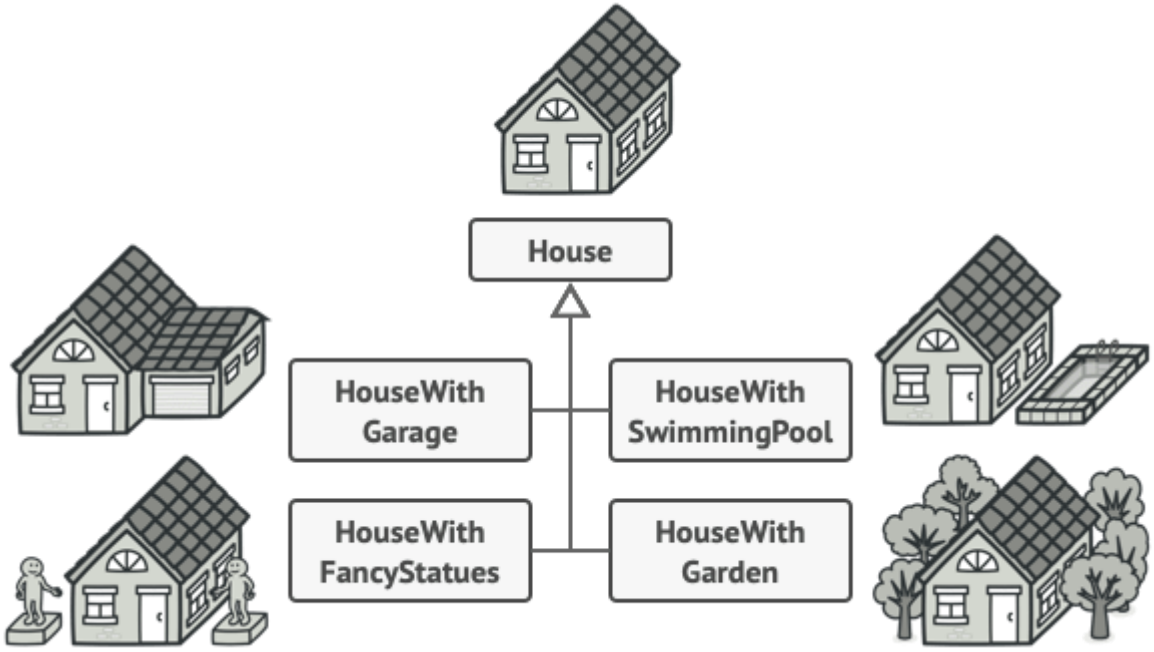
Maqsadi

Builder creational design pattern turkumiga kiruvchi patern. Bu patern yordamida murakkab obyektlar qadam baqadam qurib boriladi. Uning yordamida bir xil tuzilishdagi koddan foydalangan holda obyektning turli xildagi tiplari va ko’rinishlarini ishlab chiqishga imkon beradi.



Muammo

Ko'plab sohalarni va ichma ich joylashgan ob'ektlarni qiyin va bosqichma-bosqich initsializatsiya qilish (ishga tushirish)ni talab qiladigan murakkab ob'ektni tasavvur qiling. Kodni bunday initsializatsiya qilish odatda ko’plab kiruvchi parameterlarga ega bo’lgan “qo’rqinchli” konstruktorga ega bo’ladi. Yoki undan ham yomonroq holatda bo’lishi mumkin: butun bir mijozning kodida shunday holatda bo’lishi mumkin.

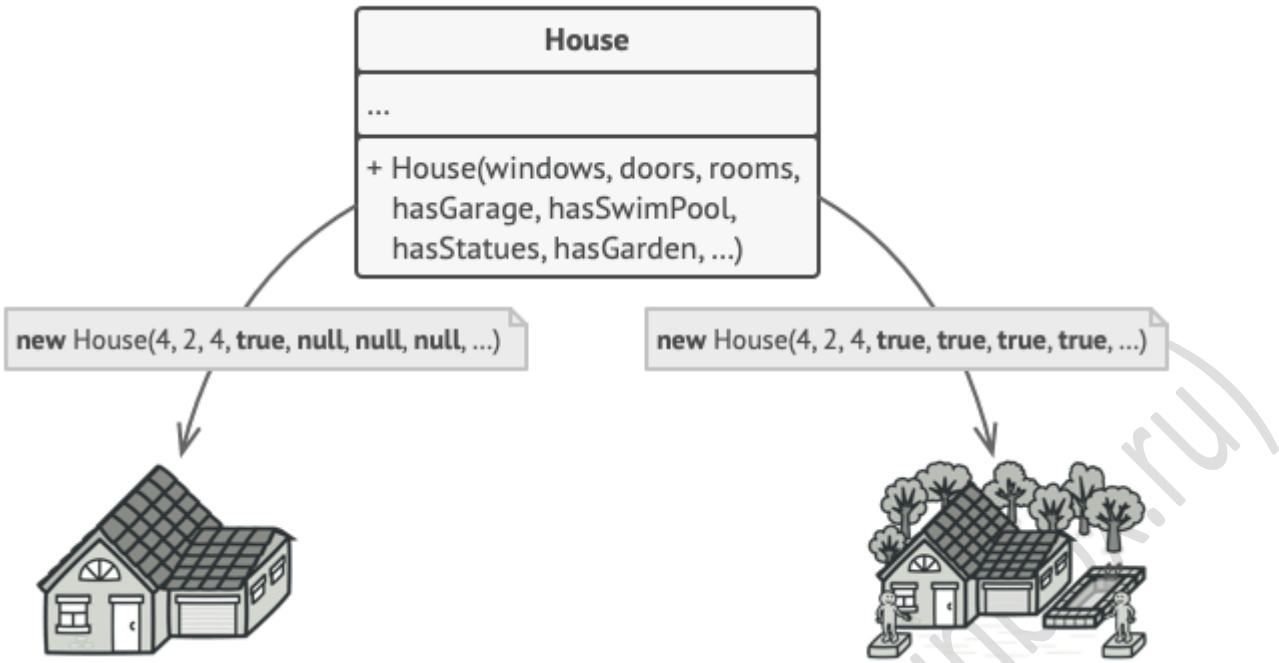


Har bir obyekt konfiguratsiyasi uchun bola klass yaratib, dasturni juda murakkablashtirib yuborishingiz mumkin

Misol uchun, *House* obyektini qanday yaratish haqida o'ylab ko'raylik. Oddiy uy qurish uchun, 4 ta devor, pol qurish kerak, eshik va deraza o'rnatish kerak, tom qilish kerak bo'ladi. Ammo, siz kattaroq, yorug'roq va hovlisi va boshqa qulayliklari (masalan, isitish tizimi) bor uy qurmoqchi bo'lsangizchi?!

Bunga eng oddiy yechim – bu House klassidan kerakli parameterlarni qamrab olgan bir nechta bola klasslarni yaratish. Ammo, oxirida, juda ko'p bola klasslarni yaratib qo'yganingizni ko'rasiz. Har qanday yangi parameter bunday klasslar sonining oshishiga olib keladi.

Bola klasslarni ko'paytirib tashlamaydigan yana boshqa yondashuv ham mavjud. Bunda uy obyektini boshqaruvchi parameterlarga ega bo'lgan katta hajmli konstruktorni asosiy House klassida to'g'ridan to'g'ri berish kerak bo'ladi. Garchi bu yondashuv bola klasslarni yo'qotsa ham, boshqa muammoni keltirib chiqaradi.

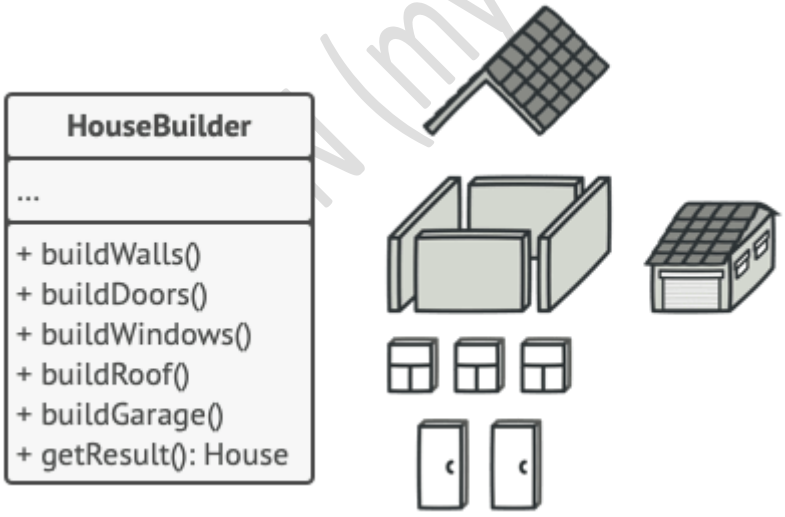


Ko'plab parameterlarga ega konstruktor o'zining kamchiliklariga ega bo'ladi: doim ham barcha parameterlar kerak bo'lavermaydi.

Ko'pgina hollarda, ko'pchilik parameterlar foydalanilmaydi. Konstruktor esa juda yomon ko'rinib qoladi. Masalan, faqat uyning bir qismidagina cho'milish hovuzi bo'lishi mumkin, natijada esa hovuzga bog'liq bo'lgan parameter 10 tadan 9 ta holatda ishlatilmaydi.

Yechim

Builder patern yordamida siz obyektning konstruktor kodini klassdan chiqarib olib, ***builder***lar deb nomlangan alohida obyektlarga o'tkazasiz.



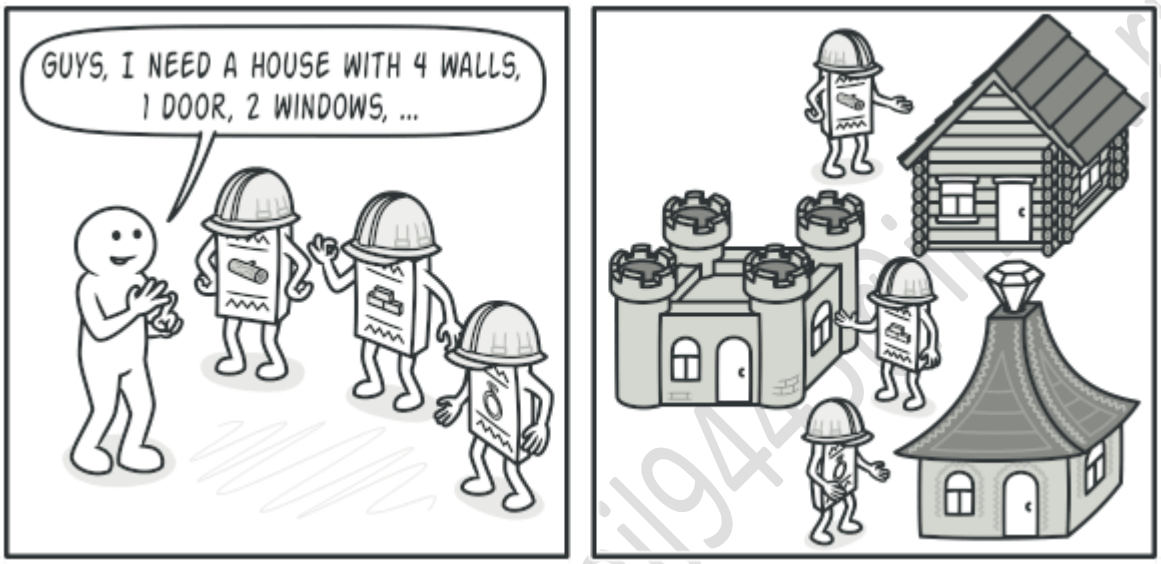
Builder paterni sizga murakkab obyektlarni qadam baqadam qurishga imkon beradi. Builder patern natijaviy mahsulot qurib bitkazilmagunicha uni boshqa obyektlar ishlatishiga yo'l qo'ymaydi.

Ushbu patern obyekt qurilishini bir qancha qadamlarda tashkillashtiradi (m: buildWalls, buildDoor). Obyektni yaratish uchun, builder obyektidagi berilgan qadamlarni ishga tushirish kerak bo'ladi. Muhim qismi shundaki,

siz barcha qadamlarni bajarishingiz shart bo’lmaydi. Masalan, sizga hovuz qismi kerak bo’lmasa, hovuzni quradigan qadamni ishlatib o’tirmaysiz.

Ba’zi paytlarda, turli ko’rinishdagi mahsulotlarni yaratishda ayrim qurilish qadamlari boshqacha ishlatilishni talab qilishi mumkin. Masalan, qaysidir xona devorini yog’ochdan qurmoqchisiz, hovlining tashqi devorlarini esa toshdan.

Bu holatda, bir xildagi qurilish qadamlarini ishlatadigan turlicha builder klasslar yaratiladi, biroq boshqacha yo’sinda yaratiladi. Keyin, siz bu builderlarni turli xil obyektlarni ishlab chiqish uchun qurilish jarayonida ishlatishingiz mumkin bo’ladi.

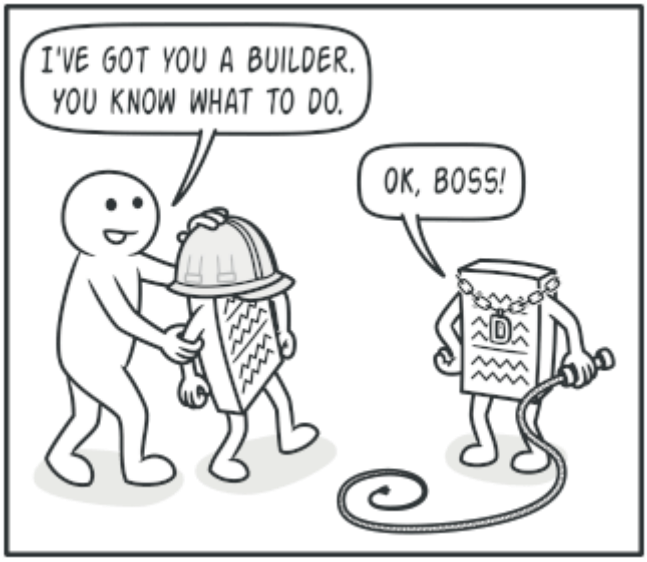


Turli xil builderlar bir xil vazifani turlich yo’lda ishga tushiradi.

Masalan, faraz qiling, bitta builder hamma narsani yog’och va shishadan, ikkinchisi esa tosh va temirdan, uchinchisi oltin va javohirlardan quradi. Bir xildagi qadamlarni ishlatib, birinchi builderdan oddiy uy, ikkinchisidan qal’a, uchinchisidan saroyni olishingiz mumkin. Biroq, bu faqat, agar qurilish qadamlarini chaqiruvchi mijoz kodi umumiy interfeysdan foydalanayotgan builderlar bilan o’zaro aloqada bo’la olsagina ishlashi mumkin.

Director

Yana davom etgan holda, yuqorida ishlatilgan qadamlar ketma-ketligini builder qadamlariga chiqarib olamiz va ulardan director deb nomlangan alohida klasslarda natijaviy mahsulotimizni qurishda foydalanamiz. Director klass qurilish qadamlarining bajarilish tartibini belgilaydi, builder esa shu qadamlarning bajarilishini ta’minlaydi.



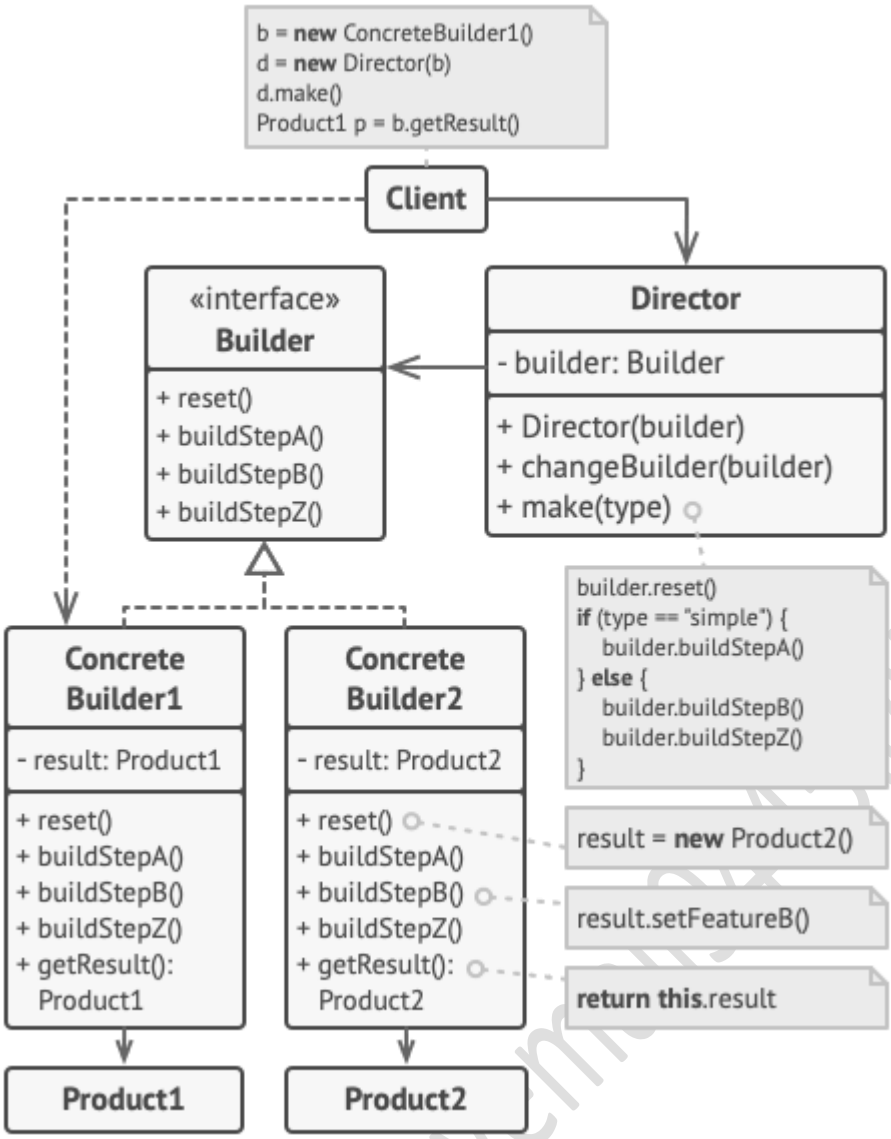
Director ishlayotgan mahsulotni olish uchun qaysi qurilish qadamini olishni biladi.

Dasturingizda director klassini bo'lishi qat'iy qilib belgilanmagan. Qurilish qadamlarini doimo berilgan ketma ketlikda to'g'ridan to'g'ri mijoz kodidan turib ham ishlatishingiz mumkin. Biroq, director klassi turli xildagi qurilish ishlarini joylashtirishga eng qulay joy bo'lishi mumkin. Shu sababli ham, siz ularni dasturingizning turli qismlarida qayta ishlatishingiz mumkin bo'ladi.

Bundan tashqari, director klass natijaviy mahsulotning qurilishi haqidagi ma'lumotlarni mijoz kodidan to'lig'icha berkitadi. Mijozga faqat builderni director bilan bog'lash, director bilan qurilishni ishga tushirish va bulderdan natijani olish kerak bo'ladi xolos.

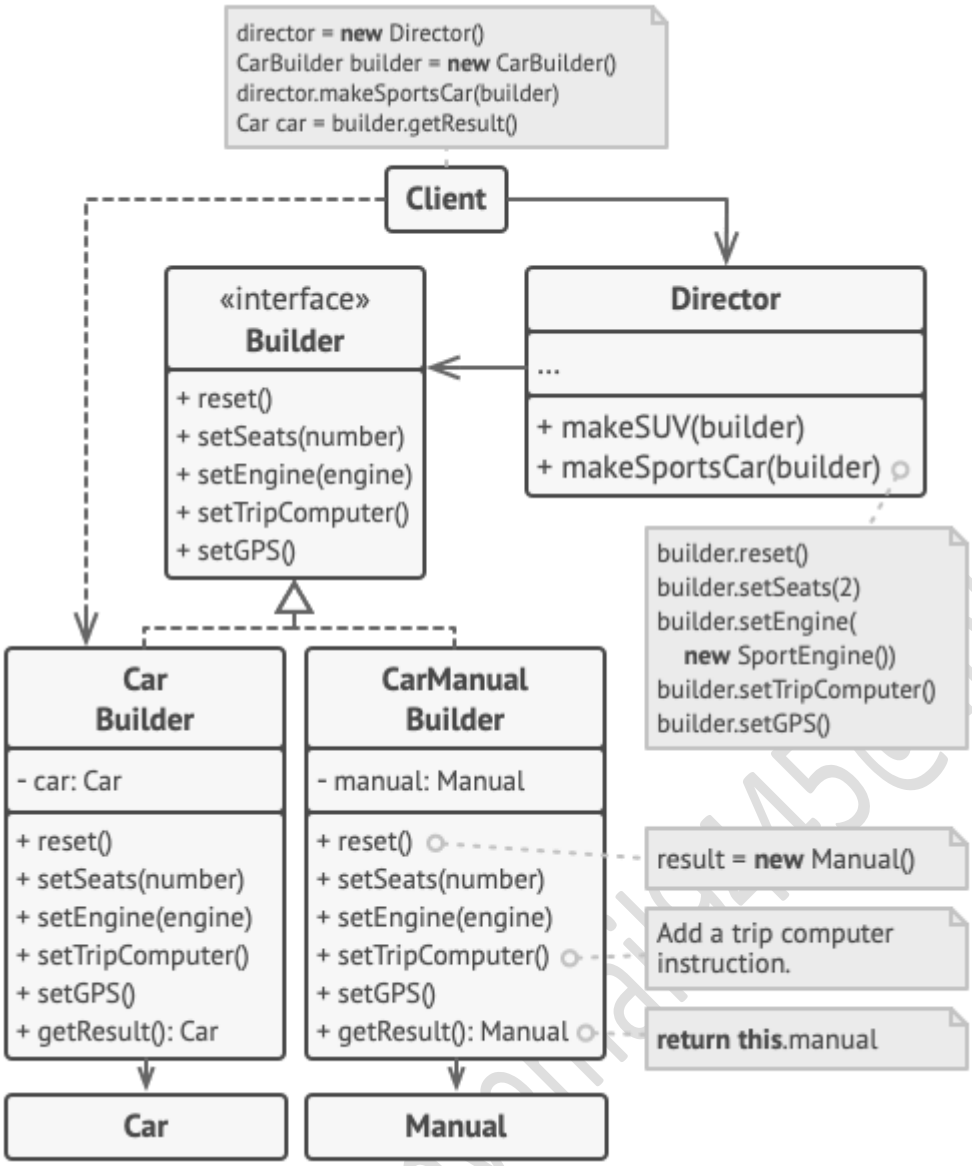
Tuzilishi

1. Builder interfeysi barcha turdagi builderlar uchun umumiy bo'lgan mahsulotni qurish qadamlarini e'lon qiladi.
2. Concrete Builderlar qurilish qadamlarining turli xilda ishlatilishini ta'minlaydi. Concrete Builderlar umumiy interfeysga bo'ysunmasdan natijaviy mahsulotlarni yaratishi mumkin.
3. Natijaviy mahsulotlar natijaviy obyektlar hisoblanadi. Turli xildagi builderlar tomonidan qurilgan mahsulotlar bir xildagi klass iyerarxiyasiga yoki interfeysga tegishli bo'lishi shart emas.
4. Director klassi qaysi qurilish qadamlarini ishlatishi tartibini belgilaydi. Shuning uchun ham siz natijaviy mahsulotlarning aniq ko'rinishdagi konfiguratsiyalarini yaratishingiz va qayta ishlatishingiz mumkin.
5. Mijoz builder obyektlaridan biri bilan directorni bog'lashi kerak. Odatda, bu director konstruktorining paramaterlari orqali faqat bir marta amalga oshiriladi. Keyin, director o'sha builder obyektini qolgan barcha konstructorlar uchun ishlatadi. Biroq, mijoz kodi directorning natijani beruvchi metodiga builder obyektni bergadigan vaziyati uchun alternativ yondashuv ham bor. Bu holatda, siz director yordamida nimadir yaratayotgan har bir paytingizda turli xildagi builderlardan foydalanishingiz mumkin.



Psevdokod

Bu yerda berilgan Builder paterni misoli mashinalar kabi turli xildagi mahsulot turlarini qurganda va ular uchun mos keluvchi qo'llanmalarni yaratganda bir xildagi obyekt konstruktsiyasi kodini qayta ishlatishni tasvirlagan.



Mashinalar va shu mashinalarga mos keluvchi foydalanuvchi qo'llanmalarini qurishning qadam baqadam misoli

Mashina yuzlab turli usullar yordamida quriladigan murakkab obyekt hisoblanadi. Katta konstructor kodiga ega Car klassini yaratish o'rniga, mashinani yig'uvchi kodlarni alohida car builder klassiga ajratamiz. Bu klass mashinaning turli qismlarini konfiguratsiyalovchi metodlar to'plamiga ega bo'ladi.

Agar mijoz kodiga mashinaning maxsus, yaxshi sozlangan modelini yig'ish kerak bo'lsa, u to'g'ridan to'g'ri builder bilan ishlaydi. Boshqa tomondan, mijoz yig'ishni turli xildagi ko'pchilik mashhur mashina modellarini qurish uchun builderdan qanday foydalanishni biladigan director klassiga ham topshirishi mumkin.

Hayron qolishingiz mumkin, lekin har bir mashinada o'zining qo'llanmasi bo'ladi (rostdanmi, kim o'qiydi uni?). Qo'llanma mashinaning har bitta xususiyatini tushuntirib beradi. Shu sababli, turli modellarda qo'llanma ma'lumoti har xil bo'ladi. Shuning uchun ham, mashinalar va ularning qo'llanmalari uchun mavjud qurilish jarayoni qayta ishlatiladi. Albatta, qo'llanmani qurish mashina qurish bilan bir xil bo'lmaydi, shuning sababli ham, biz qo'llanmani yaratishga mo'ljallangan boshqa builder klassni berishimiz kerak. Ushbu klass o'zining mashina quruvchi boshqa klasslar singari bir xildagi qurish metodlaridan foydalanadi, lekin mashina yasash o'rniga ularni tavsiflab beradi. Bu builderlarni bir xildagi director obyektiga berish orqali biz yoki mashina yoki qo'llanma qurishimiz mumkin bo'ladi.

Yakuniy qismda, natijaviy obyektni olamiz. Garchi o’zaro bog’langan bo’lsa ham, metal mashina va qog’oz qo’llanma umuman farqli narsalar hisoblanadi. Aniq (concrete) mahsulot klassga directorni bog’lamay turib natijani beruvchi metodni directorda joylashtira olmaymiz. Shu sababli, ishni bajaradigan builderdan qurilish natijasini olamiz.

```
// Using the Builder pattern makes sense only when your products
// are quite complex and require extensive configuration. The
// following two products are related, although they don't have
// a common interface.

class Car is
    // A car can have a GPS, trip computer and some number of
    // seats. Different models of cars (sports car, SUV,
    // cabriolet) might have different features installed or
    // enabled.

class Manual is
    // Each car should have a user manual that corresponds to
    // the car's configuration and describes all its features.


// The builder interface specifies methods for creating the
// different parts of the product objects.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// The concrete builder classes follow the builder interface and
// provide specific implementations of the building steps. Your
// program may have several variations of builders, each
// implemented differently.
class CarBuilder implements Builder is
    private field car:Car

    // A fresh builder instance should contain a blank product
    // object which it uses in further assembly.
    constructor CarBuilder() is
        this.reset()

    // The reset method clears the object being built.
    method reset() is
        this.car = new Car()

    // All production steps work with the same product instance.
    method setSeats(...) is
        // Set the number of seats in the car.

    method setEngine(...) is
        // Install a given engine.

    method setTripComputer(...) is
        // Install a trip computer.

    method setGPS(...) is
        // Install a global positioning system.

    // Concrete builders are supposed to provide their own
    // methods for retrieving results. That's because various
```

```
    // types of builders may create entirely different products
    // that don't all follow the same interface. Therefore such
    // methods can't be declared in the builder interface (at
    // least not in a statically-typed programming language).
    //
    // Usually, after returning the end result to the client, a
    // builder instance is expected to be ready to start
    // producing another product. That's why it's a usual
    // practice to call the reset method at the end of the
    // `getProduct` method body. However, this behavior isn't
    // mandatory, and you can make your builder wait for an
    // explicit reset call from the client code before disposing
    // of the previous result.
    method getProduct():Car is
        product = this.car
        this.reset()
        return product

// Unlike other creational patterns, builder lets you construct
// products that don't follow the common interface.
class CarManualBuilder implements Builder is
    private field manual:Manual

    constructor CarManualBuilder() is
        this.reset()

    method reset() is
        this.manual = new Manual()

    method setSeats(...) is
        // Document car seat features.

    method setEngine(...) is
        // Add engine instructions.

    method setTripComputer(...) is
        // Add trip computer instructions.

    method setGPS(...) is
        // Add GPS instructions.

    method getProduct():Manual is
        // Return the manual and reset the builder.


// The director is only responsible for executing the building
// steps in a particular sequence. It's helpful when producing
// products according to a specific order or configuration.
// Strictly speaking, the director class is optional, since the
// client can control builders directly.
class Director is
    private field builder:Builder

    // The director works with any builder instance that the
    // client code passes to it. This way, the client code may
    // alter the final type of the newly assembled product.
    method setBuilder(builder:Builder)
        this.builder = builder

    // The director can construct several product variations
    // using the same building steps.
    method constructSportsCar(builder: Builder) is
```

```
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

    method constructSUV(builder: Builder) is
        // ...


// The client code creates a builder object, passes it to the
// director and then initiates the construction process. The end
// result is retrieved from the builder object.
class Application is

    method makeCar() is
        director = new Director()

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

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

        // The final product is often retrieved from a builder
        // object since the director isn't aware of and not
        // dependent on concrete builders and products.
        Manual manual = builder.getProduct()
```

Amaliy ishlashi

Builder paterndan

- **“teleskopik konstruktor”dan qutulish uchun.**
- **Builder paterndan ayrim mahsulotlarning turli ko’rinishlarini yartishda.**
- **Builder paterndan Composite daraxtlarni yoki boshqa murakkab obyektlarni qurishda.**

foydalaning

Ishlatilishi.

1. Barja mavjud mahsulot ko’rinishlarini qurish uchun umumiy qurilish qadamlarini aniq belgilay olishingizga amin bo’ling. Aks holda, paternni ishlata olmasligingiz mumkin.
2. Bu qadamlarni asosiy builder interfeysida e’lon qiling.
3. Har bir mahsulot ko’rinishi uchun concrete builder klass yarating va ularning qurilish qadamlarini ishlating.
   Qurilish natijasini olish uchun metod ishlatishni unutmang. Bu metodning builder interfeysida e’lon qilinmasligiga sabab, turli xil builderlar umumiy interfeysga ega bo’lmagan mahsulotlarni yaratishi mumkinligida. Shuning uchun, siz bu metodning qanday tip qaytarishini bilmaysiz. Biroq, agar bitta iyerarxiyadagi mahsulotlar bilan shug’ullanayotgan bo’lsangiz, natijani beruvchi metodni asosiy interfeysga qo’shsangiz bo’ladi.
4. Direct klass yaratish haqida o’ylab ko’ring. U bir xildagi builder obyektidan foydalanib mahsulot yaratishning har xil yo’llarini enkapsulyatsiyalaydi.
5. Mijoz kodi ham builder ham director obyektlarini yaratadi. Yaratish boshlanishidan oldin, mijoz directorga builder obuyektini berishi kerak bo’ladi. Odatda, mijoz buni director konstruktorining

paramterlari orqali bir marta bajaradi. Director builder obyektini qolgan barcha konstruksiyalarda ishlatadi.

6. Konstruksiya natijasini, agar barcha mahsulotlar bir xildagi interfeysni ishlatsa, to'g'ridan to'g'ri directordan olish mumkin.

Afzallik va kamchiliklari

Afzalliklari	Kamchiliklari
Obyektni qadam baqadam qurish mumkin. Qurish qadamlarini kechiktirish yoki qadamlarni rekursiv ishlatish mumkin.	Kodining umumiy murakkabligi oshadi, chunki patern ko'plab yangi klasslarni yaratishni talab qiladi.
Mahsulotlarning turli xildagi ko'rinishini yaratishda bir xildagi konstruksiya kodini qayta ishlatish mumkin.	
Yagona Javobgarlik Tamoyili. Murakkab konstruksiya kodini mahsulotning biznes logikasidan izolyatsiyalashingiz mumkin.	

Boshqa paternlarga bog'liqligi

- Ko'plab loyihalashlar Factory Method ni ishlatishdan boshlanadi va Abstract Factory, Prototype va Builder paternlari tomon rivojlanib boradi.
- Builder patern qadam baqadam murakkab obyektlarni qurishga e'tibor qaratadi. Abstract Factory paterni esa o'zaro bog'langan obyektlar oilasini yaratishga mo'ljallangan. Abstract Factory natijaviy mahsulotni darhol qaytaradi, Builder esa mahsulotni olishdan avval qo'shimcha konstruksiya qadamlarini ishlatishga imkon beradi.
- Builder paterndan murakkab Composite daraxtlarni yaratishda foydalanishingiz mumkin, chunki uning konstruksiya qadamlarini rekursiv ishlashini dasturlash mumkin.
- Abstract Factory, Builder va Prototypelarning hammasi Singletonlar sifatida ishlatilishi mumkin.

Namuna Kod

```php
<?php

namespace RefactoringGuru\Builder\Conceptual;

/**
 * The Builder interface specifies methods for creating the different parts of
 * the Product objects.
 */
interface Builder
{
    public function producePartA(): void;
```

```
    public function producePartB(): void;

    public function producePartC(): void;
}

/**
 * The Concrete Builder classes follow the Builder interface and provide
 * specific implementations of the building steps. Your program may have several
 * variations of Builders, implemented differently.
 */
class ConcreteBuilder1 implements Builder
{
    private $product;

    /**
     * A fresh builder instance should contain a blank product object, which is
     * used in further assembly.
     */
    public function __construct()
    {
        $this->reset();
    }

    public function reset(): void
    {
        $this->product = new Product1();
    }

    /**
     * All production steps work with the same product instance.
     */
    public function producePartA(): void
    {
        $this->product->parts[] = "PartA1";
    }

    public function producePartB(): void
    {
        $this->product->parts[] = "PartB1";
    }

    public function producePartC(): void
    {
        $this->product->parts[] = "PartC1";
    }

    /**
     * Concrete Builders are supposed to provide their own methods for
     * retrieving results. That's because various types of builders may create
     * entirely different products that don't follow the same interface.
     * Therefore, such methods cannot be declared in the base Builder interface
     * (at least in a statically typed programming language). Note that PHP is a
     * dynamically typed language and this method CAN be in the base interface.
     * However, we won't declare it there for the sake of clarity.
     *
     * Usually, after returning the end result to the client, a builder instance
     * is expected to be ready to start producing another product. That's why
     * it's a usual practice to call the reset method at the end of the
     * `getProduct` method body. However, this behavior is not mandatory, and
     * you can make your builders wait for an explicit reset call from the
     * client code before disposing of the previous result.
     */
```

```php
    public function getProduct(): Product1
    {
        $result = $this->product;
        $this->reset();

        return $result;
    }
}

/**
 * It makes sense to use the Builder pattern only when your products are quite
 * complex and require extensive configuration.
 *
 * Unlike in other creational patterns, different concrete builders can produce
 * unrelated products. In other words, results of various builders may not
 * always follow the same interface.
 */
class Product1
{
    public $parts = [];

    public function listParts(): void
    {
        echo "Product parts: " . implode(', ', $this->parts) . "\n\n";
    }
}

/**
 * The Director is only responsible for executing the building steps in a
 * particular sequence. It is helpful when producing products according to a
 * specific order or configuration. Strictly speaking, the Director class is
 * optional, since the client can control builders directly.
 */
class Director
{
    /**
     * @var Builder
     */
    private $builder;

    /**
     * The Director works with any builder instance that the client code passes
     * to it. This way, the client code may alter the final type of the newly
     * assembled product.
     */
    public function setBuilder(Builder $builder): void
    {
        $this->builder = $builder;
    }

    /**
     * The Director can construct several product variations using the same
     * building steps.
     */
    public function buildMinimalViableProduct(): void
    {
        $this->builder->producePartA();
    }

    public function buildFullFeaturedProduct(): void
    {
        $this->builder->producePartA();
```

```
        $this->builder->producePartB();
        $this->builder->producePartC();
    }
}

/**
 * The client code creates a builder object, passes it to the director and then
 * initiates the construction process. The end result is retrieved from the
 * builder object.
 */
function clientCode(Director $director)
{
    $builder = new ConcreteBuilder1();
    $director->setBuilder($builder);

    echo "Standard basic product:\n";
    $director->buildMinimalViableProduct();
    $builder->getProduct()->listParts();

    echo "Standard full featured product:\n";
    $director->buildFullFeaturedProduct();
    $builder->getProduct()->listParts();

    // Remember, the Builder pattern can be used without a Director class.
    echo "Custom product:\n";
    $builder->producePartA();
    $builder->producePartC();
    $builder->getProduct()->listParts();
}

$director = new Director();
clientCode($director);
```

```
Standard basic product:
Product parts: PartA1

Standard full featured product:
Product parts: PartA1, PartB1, PartC1

Custom product:
Product parts: PartA1, PartC1
```