refactoring.guru

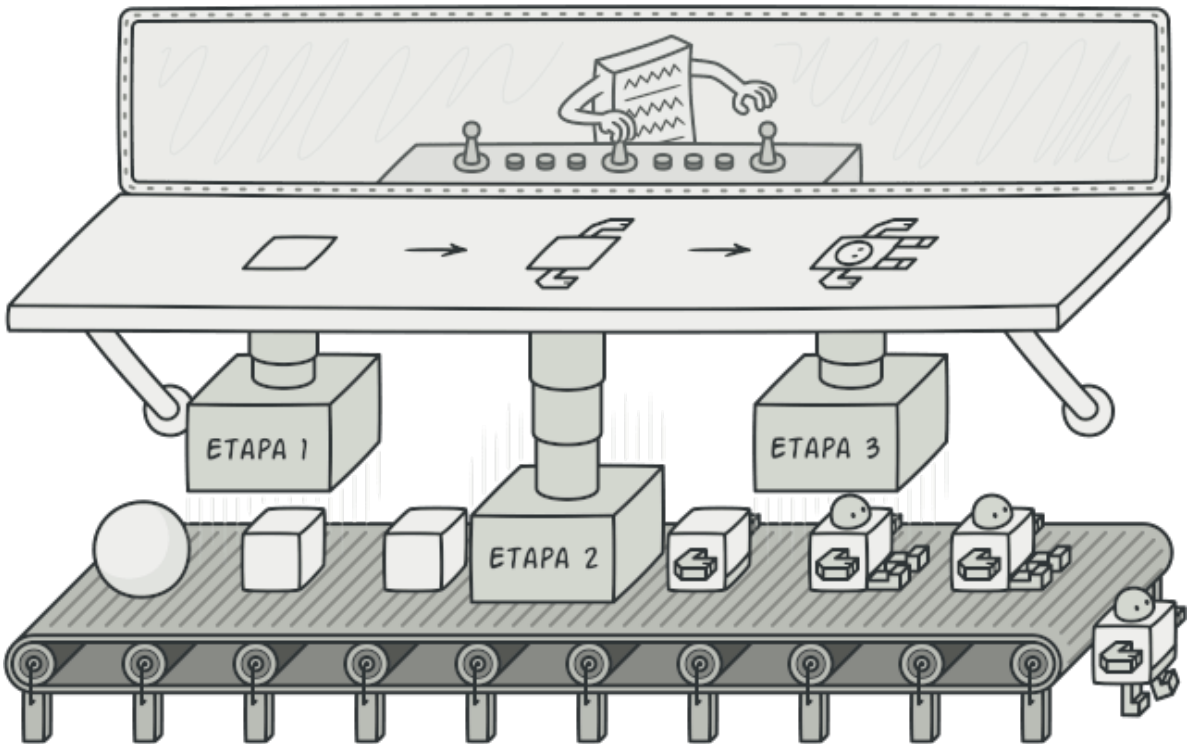
Builder

18–24 minutes

Também conhecido como: Construtor

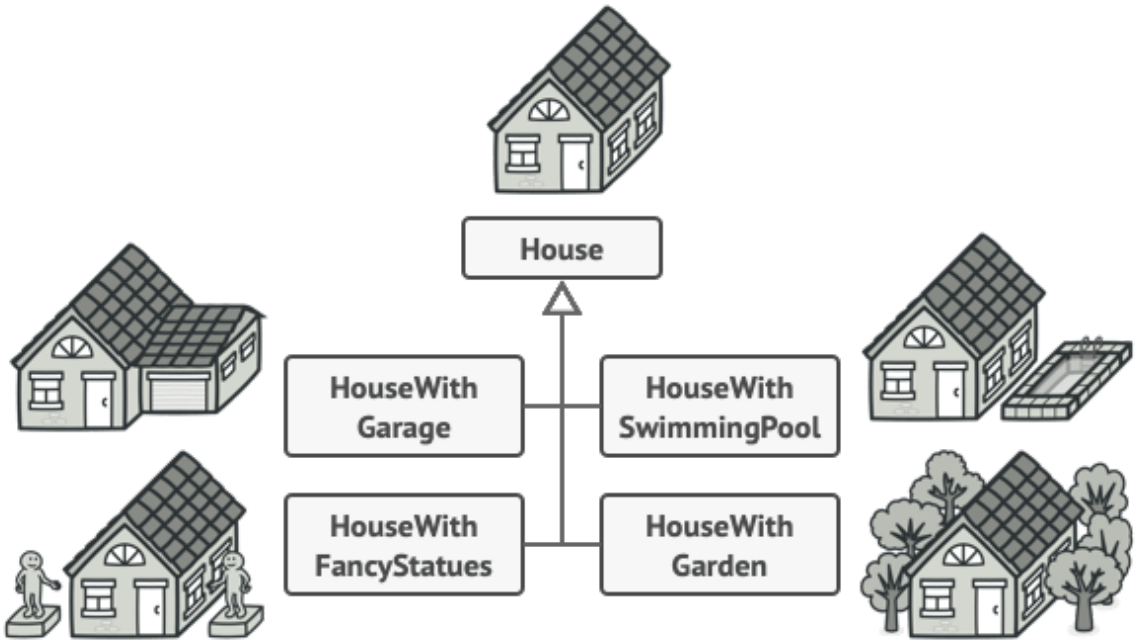
Propósito

O **Builder** é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.



Problema

Imagine um objeto complexo que necessite de uma inicialização passo a passo trabalhosa de muitos campos e objetos agrupados. Tal código de inicialização fica geralmente enterrado dentro de um construtor monstruoso com vários parâmetros. Ou pior: espalhado por todo o código cliente.



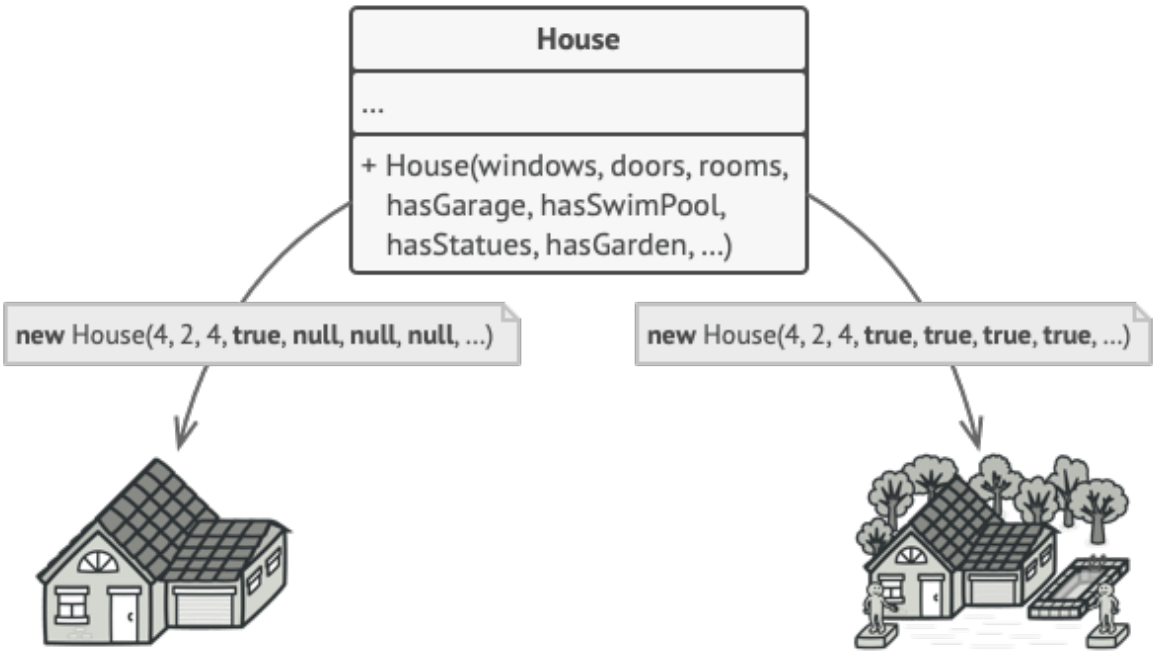
Você pode tornar o programa muito complexo ao criar subclasses para cada possível configuração de um objeto.

Por exemplo, vamos pensar sobre como criar um objeto `Casa`. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas, e construir um teto. Mas e se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como um sistema de aquecimento, encanamento, e fiação elétrica)?

A solução mais simples é estender a classe base `Casa` e criar um conjunto de subclasses para cobrir todas as combinações de parâmetros. Mas eventualmente você acabará com um número considerável de subclasses. Qualquer novo parâmetro, tal como o

estilo do pórtico, irá forçá-lo a aumentar essa hierarquia cada vez mais.

Há outra abordagem que não envolve a propagação de subclasses. Você pode criar um construtor gigante diretamente na classe `Casa` base com todos os possíveis parâmetros que controlam o objeto casa. Embora essa abordagem realmente elimine a necessidade de subclasses, ela cria outro problema.



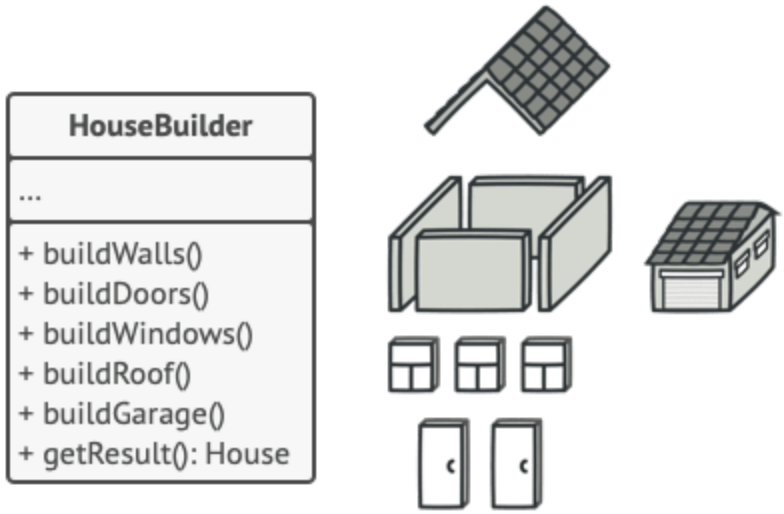
O construtor com vários parâmetros tem um lado ruim: nem todos os parâmetros são necessários todas as vezes.

Na maioria dos casos a maioria dos parâmetros não será usada, tornando as chamadas do construtor em algo feio de se ver. Por exemplo, apenas algumas casas têm piscinas, então os parâmetros relacionados a piscinas serão inúteis nove em cada dez vezes.

Solução

O padrão Builder sugere que você extraia o código de construção

do objeto para fora de sua própria classe e mova ele para objetos separados chamados *builders*. “Builder” significa “construtor”, mas não usaremos essa palavra para evitar confusão com os construtores de classe.



O padrão Builder permite que você construa objetos complexos passo a passo. O Builder não permite que outros objetos acessem o produto enquanto ele está sendo construído.

O padrão organiza a construção de objetos em uma série de etapas (`construirParedes`, `construirPorta`, etc.). Para criar um objeto você executa uma série de etapas em um objeto builder. A parte importante é que você não precisa chamar todas as etapas. Você chama apenas aquelas etapas que são necessárias para a produção de uma configuração específica de um objeto.

Algumas das etapas de construção podem necessitar de implementações diferentes quando você precisa construir várias representações do produto. Por exemplo, paredes de uma cabana podem ser construídas com madeira, mas paredes de um castelo devem ser construídas com pedra.

Nesse caso, você pode criar diferentes classes construturas que implementam as mesmas etapas de construção, mas de maneira diferente. Então você pode usar esses builders no processo de construção (i.e, um pedido ordenado de chamadas para as etapas de construção) para produzir diferentes tipos de objetos.

Builders diferentes executam a mesma tarefa de várias maneiras.

Por exemplo, imagine um builder que constrói tudo de madeira e vidro, um segundo builder que constrói tudo com pedra e ferro, e um terceiro que usa ouro e diamantes. Ao chamar o mesmo conjunto de etapas, você obtém uma casa normal do primeiro builder, um pequeno castelo do segundo, e um palácio do terceiro. Contudo, isso só vai funcionar se o código cliente que chama as etapas de construção é capaz de interagir com os builders usando uma interface comum.

Diretor

Você pode ir além e extrair uma série de chamadas para as etapas do builder que você usa para construir um produto em uma classe separada chamada *diretor*. A classe diretor define a ordem

na qual executar as etapas de construção, enquanto que o builder provê a implementação dessas etapas.

O diretor sabe quais etapas de construção executar para obter um produto que funciona.

Ter uma classe diretor em seu programa não é estritamente necessário. Você sempre pode chamar as etapas de construção em uma ordem específica diretamente do código cliente. Contudo, a classe diretor pode ser um bom lugar para colocar várias rotinas de construção para que você possa reutilizá-las em qualquer lugar do seu programa.

Além disso, a classe diretor esconde completamente os detalhes da construção do produto do código cliente. O cliente só precisa associar um builder com um diretor, inicializar a construção com o diretor, e então obter o resultado do builder.

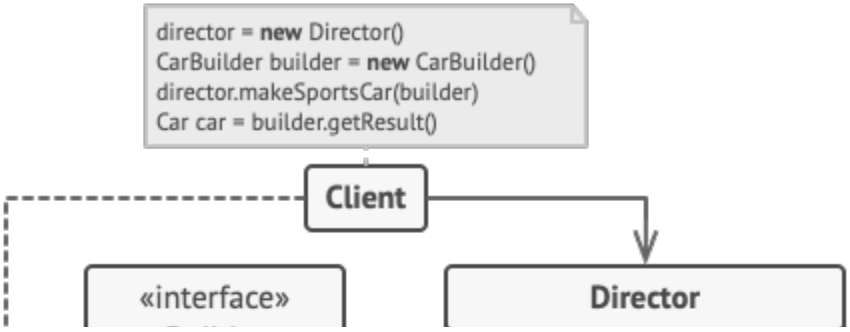
Estrutura

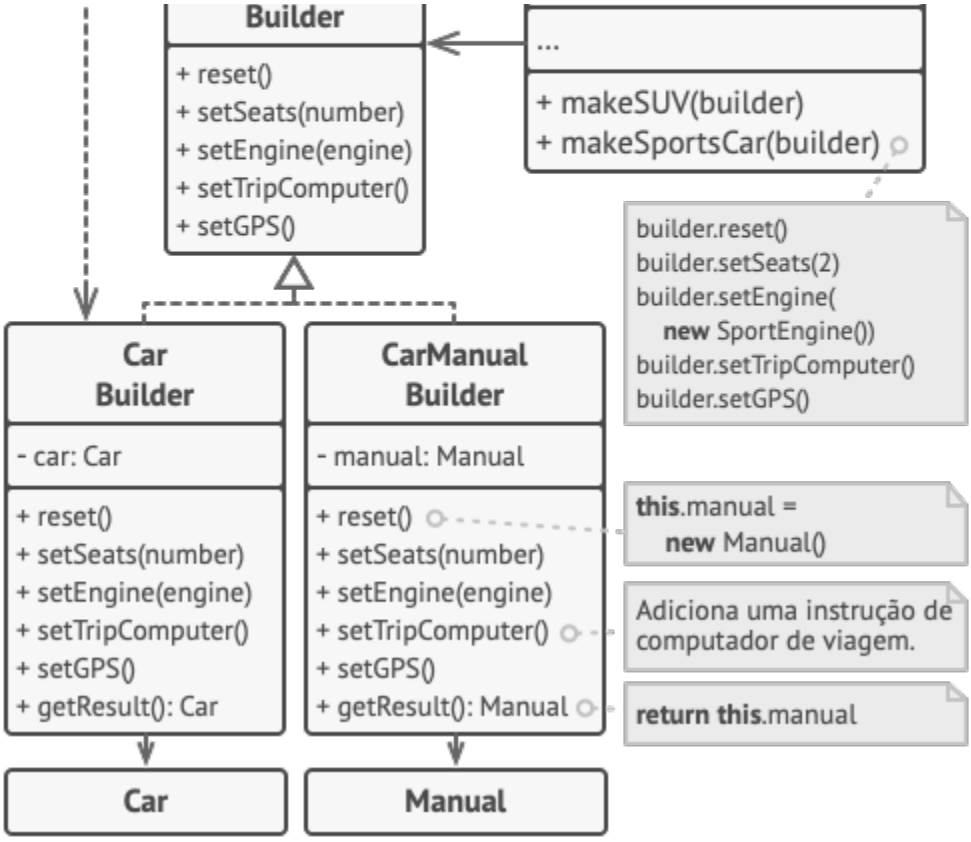
1. A interface **Builder** declara etapas de construção do produto que são comuns a todos os tipos de builders.

2. **Builders Concretos** provém diferentes implementações das etapas de construção. Builders concretos podem produzir produtos que não seguem a interface comum.
3. **Produtos** são os objetos resultantes. Produtos construídos por diferentes builders não precisam pertencer a mesma interface ou hierarquia da classe.
4. A classe **Diretor** define a ordem na qual as etapas de construção são chamadas, então você pode criar e reutilizar configurações específicas de produtos.
5. O **Cliente** deve associar um dos objetos builders com o diretor. Usualmente isso é feito apenas uma vez, através de parâmetros do construtor do diretor. O diretor então usa aquele objeto builder para todas as futuras construções. Contudo, há uma abordagem alternativa para quando o cliente passa o objeto builder ao método de produção do diretor. Nesse caso, você pode usar um builder diferente a cada vez que você produzir alguma coisa com o diretor.

Pseudocódigo

Este exemplo do padrão **Builder** ilustra como você pode reutilizar o mesmo código de construção de objeto quando construindo diferentes tipos de produtos, tais como carros, e como criar manuais correspondentes para eles.





O exemplo do passo a passo da construção de carros e do manual do usuário que se adapta a aqueles modelos de carros.

Um carro é um objeto complexo que pode ser construído em centenas de maneiras diferentes. Ao invés de inchar a classe `Carro` com um construtor enorme, nós extraímos o código de montagem do carro em uma classe de construção de carro separada. Essa classe tem um conjunto de métodos para configurar as várias partes de um carro.

Se o código cliente precisa montar um modelo de carro especial e ajustado, ele pode trabalhar com o builder diretamente. Por outro lado, o cliente pode delegar a montagem à classe diretor, que sabe como usar um builder para construir diversos modelos de carros populares.

Você pode ficar chocado, mas cada carro precisa de um manual (sério, quem lê aquilo?). O manual descreve cada funcionalidade

do carro, então os detalhes dos manuais variam de acordo com os diferentes modelos. É por isso que faz sentido reutilizar um processo de construção existente para ambos os carros e seus respectivos manuais. É claro, construir um manual não é o mesmo que construir um carro, e é por isso que devemos providenciar outra classe builder que se especializa em compor manuais. Essa classe implementa os mesmos métodos de construção que seus parentes builder de carros, mas ao invés de construir partes de carros, ela as descreve. Ao passar esses builders ao mesmo objeto diretor, podemos tanto construir um carro como um manual.

A parte final é obter o objeto resultante. Um carro de metal e um manual de papel, embora relacionados, são coisas muito diferentes. Não podemos colocar um método para obter resultados no diretor sem ligar o diretor às classes de produto concretas. Portanto, nós obtemos o resultado da construção a partir do builder que fez o trabalho.

```
// Usar o padrão Builder só faz sentido quando seus produtos são
// bem complexos e requerem configuração extensiva. Os dois
// produtos a seguir são relacionados, embora eles não tenham
// uma interface em comum.
class Car is
    // Um carro pode ter um GPS, computador de bordo, e alguns
    // assentos. Diferentes modelos de carros (esportivo, SUV,
    // conversível) podem ter diferentes funcionalidades
    // instaladas ou equipadas.

class Manual is
    // Cada carro deve ter um manual do usuário que corresponda
    // a configuração do carro e descreva todas suas
```

```
    // funcionalidades.


// A interface builder especifica métodos para criar as
// diferentes partes de objetos produto.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// As classes builder concretas seguem a interface do
// builder e fornecem implementações específicas das etapas
// de construção. Seu programa pode ter algumas variações de
// builders, cada uma implementada de forma diferente.
class CarBuilder implements Builder is
    private field car:Car

    // Uma instância fresca do builder deve conter um objeto
    // produto em branco na qual ela usa para montagem futura.
    constructor CarBuilder() is
        this.reset()

    // O método reset limpa o objeto sendo construído.
    method reset() is
        this.car = new Car()

    // Todas as etapas de produção trabalham com a mesma
    // instância de produto.
```

```
method setSeats(...) is
    // Define o número de assentos no carro.

method setEngine(...) is
    // Instala um tipo de motor.

method setTripComputer(...) is
    // Instala um computador de bordo.

method setGPS(...) is
    // Instala um sistema de posicionamento global.

// Builders concretos devem fornecer seus próprios
// métodos para recuperar os resultados. Isso é porque
// vários tipos de builders podem criar produtos
// inteiramente diferentes que nem sempre seguem a mesma
// interface. Portanto, tais métodos não podem ser
// declarados na interface do builder (ao menos não em
// uma linguagem de programação de tipo estático).
//
// Geralmente, após retornar o resultado final para o
// cliente, espera-se que uma instância de builder comece
// a produzir outro produto. É por isso que é uma prática
// comum chamar o método reset no final do corpo do método
// `getProduct`. Contudo este comportamento não é
// obrigatório, e você pode fazer seu builder esperar por
// uma chamada explícita do reset a partir do código cliente
// antes de se livrar de seu resultado anterior.
method getProduct():Car is
    product = this.car
```

```
        this.reset()
        return product

// Ao contrário dos outros padrões criacionais, o Builder
// permite que você construa produtos que não seguem uma
// interface comum.
class CarManualBuilder implements Builder is
    private field manual:Manual

    constructor CarManualBuilder() is
        this.reset()

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

    method setSeats(...) is
        // Documenta as funcionalidades do assento do carro.

    method setEngine(...) is
        // Adiciona instruções do motor.

    method setTripComputer(...) is
        // Adiciona instruções do computador de bordo.

    method setGPS(...) is
        // Adiciona instruções do GPS.

    method getProduct():Manual is
        // Retorna o manual e reseta o builder.
```

```
// O diretor é apenas responsável por executar as etapas de
// construção em uma sequência em particular. Isso ajuda quando
// produzindo produtos de acordo com uma ordem específica ou
// configuração. A rigor, a classe diretor é opcional, já que o
// cliente pode controlar os builders diretamente.
class Director is
    // O diretor trabalha com qualquer instância builder que
    // o código cliente passar a ele. Dessa forma, o código
    // cliente pode alterar o tipo final do produto recém
    // montado. O diretor pode construir diversas variações
    // do produto usando as mesmas etapas de construção.
    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



// O código cliente cria um objeto builder, passa ele para o
// diretor e então inicia o processo de construção. O resultado
// final é recuperado do objeto builder.
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)

// O produto final é frequentemente retornado de um
// objeto builder uma vez que o diretor não está
// ciente e não é dependente de builders e produtos
// concretos.
Manual manual = builder.getProduct()
```

Aplicabilidade

Use o padrão Builder para se livrar de um “construtor telescópico”.

Digamos que você tenha um construtor com dez parâmetros opcionais. Chamar um monstro desses é muito inconveniente; portanto, você sobrecarrega o construtor e cria diversas versões curtas com menos parâmetros. Esses construtores ainda se referem ao principal, passando alguns valores padrão para qualquer parâmetro omitido.

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

Criar tal monstro só é possível em linguagens que suportam

sobrecarregamento de método, tais como C# ou Java.

O padrão Builder permite que você construa objetos passo a passo, usando apenas aquelas etapas que você realmente precisa. Após implementar o padrão, você não vai mais precisar amontoar dúzias de parâmetros em seus construtores.

Use o padrão Builder quando você quer que seu código seja capaz de criar diferentes representações do mesmo produto (por exemplo, casas de pedra e madeira).

O padrão Builder pode ser aplicado quando a construção de várias representações do produto envolvem etapas similares que diferem apenas nos detalhes.

A interface base do builder define todas as etapas de construção possíveis, e os buildrs concretos implementam essas etapas para construir representações particulares do produto. Enquanto isso, a classe diretor guia a ordem de construção.

Use o Builder para construir árvores Composite ou outros objetos complexos.

O padrão Builder permite que você construa produtos passo a passo. Você pode adiar a execução de algumas etapas sem quebrar o produto final. Você pode até chamar etapas recursivamente, o que é bem útil quando você precisa construir uma árvore de objetos.

Um builder não expõe o produto não finalizado enquanto o processo de construção estiver executando etapas. Isso previne o código cliente de obter um resultado incompleto.

Como implementar

1. Certifique-se que você pode definir claramente as etapas comuns de construção para construir todas as representações do produto disponíveis. Do contrário, você não será capaz de implementar o padrão.
2. Declare essas etapas na interface builder base.
3. Crie uma classe builder concreta para cada representação do produto e implemente suas etapas de construção.

   Não se esqueça de implementar um método para recuperar os resultados da construção. O motivo pelo qual esse método não pode ser declarado dentro da interface do builder é porque vários builders podem construir produtos que não tem uma interface comum. Portanto, você não sabe qual será o tipo de retorno para tal método. Contudo, se você está lidando com produtos de uma única hierarquia, o método de obtenção pode ser adicionado com segurança para a interface base.
4. Pense em criar uma classe diretor. Ela pode encapsular várias maneiras de construir um produto usando o mesmo objeto builder.
5. O código cliente cria tanto os objetos do builder como do diretor. Antes da construção começar, o cliente deve passar um objeto builder para o diretor. Geralmente o cliente faz isso apenas uma vez, através de parâmetros do construtor do diretor. O diretor usa o objeto builder em todas as construções futuras. Existe uma alternativa onde o builder é passado diretamente ao método de construção do diretor.
6. O resultado da construção pode ser obtido diretamente do diretor apenas se todos os produtos seguirem a mesma interface. Do contrário o cliente deve obter o resultado do builder.

Prós e contras

- Você pode construir objetos passo a passo, adiar as etapas de construção ou rodar etapas recursivamente.
- Você pode reutilizar o mesmo código de construção quando construindo várias representações de produtos.
- *Princípio de responsabilidade única*. Você pode isolar um código de construção complexo da lógica de negócio do produto.
- A complexidade geral do código aumenta uma vez que o padrão exige criar múltiplas classes novas.

Relações com outros padrões

- Muitos projetos começam usando o Factory Method (menos complicado e mais customizável através de subclasses) e evoluem para o Abstract Factory, Prototype, ou Builder (mais flexíveis, mas mais complicados).
- O Builder foca em construir objetos complexos passo a passo. O Abstract Factory se especializa em criar famílias de objetos relacionados. O *Abstract Factory* retorna o produto imediatamente, enquanto que o *Builder* permite que você execute algumas etapas de construção antes de buscar o produto.
- Você pode usar o Builder quando criar árvores Composite complexas porque você pode programar suas etapas de construção para trabalhar recursivamente.
- Você pode combinar o Builder com o Bridge: a classe *diretor* tem um papel de abstração, enquanto que diferentes *construtores* agem como *implementações*.

- As Fábricas Abstratas, Construtores, e Protótipos podem todos ser implementados como Singletons.