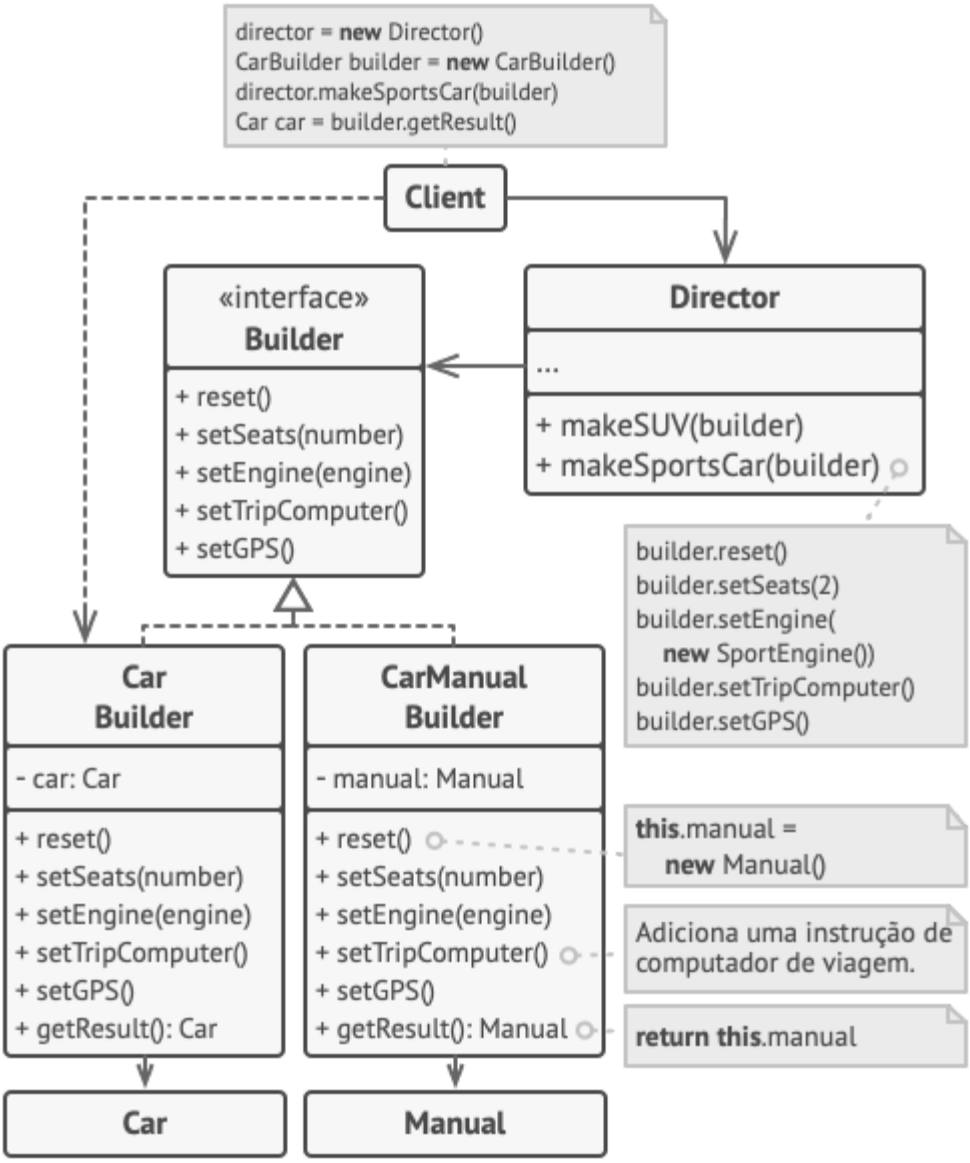
Builder

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.

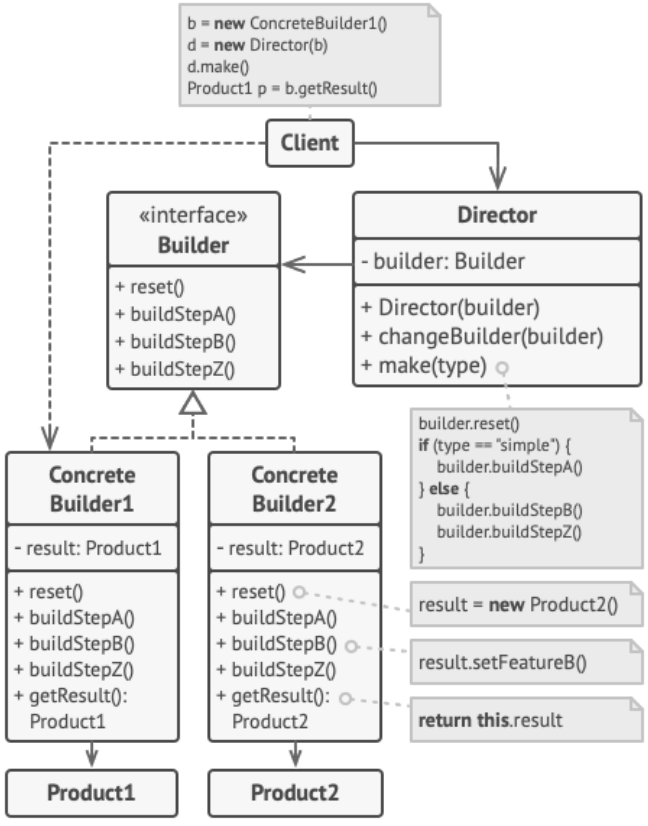
```
// Usar o padrão Builder só faz sentido quando seus produtos são
// bem complexos e requerem configuração extensiva. Os dois
// produtos a seguir são relacionados, embora eles não tenham
// uma interface em comum.
class Car is
    // Um carro pode ter um GPS, computador de bordo, e alguns
    // 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
    private field builder:Builder

    // 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.
    method setBuilder(builder:Builder)
        this.builder = builder

    // 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()
```

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.

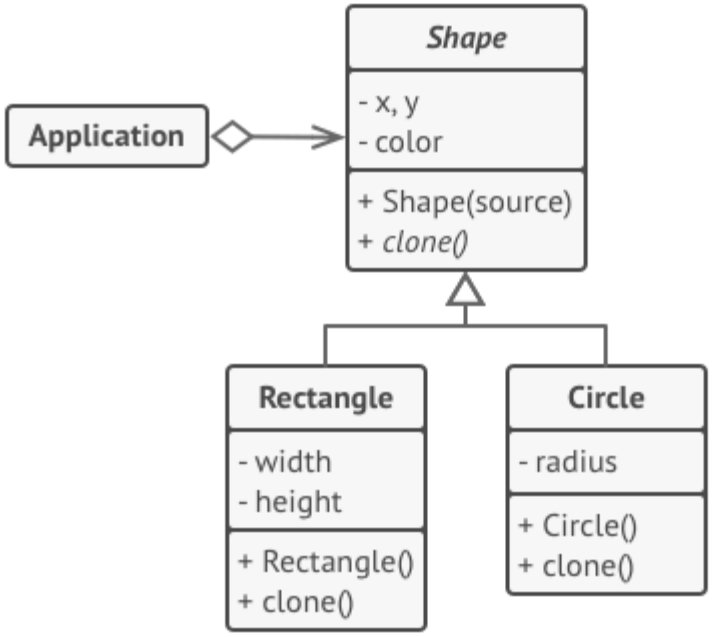
4. A classe Diretor define a ordem na qual as etapas de construção são chamadas, então você pode criar e reutilizar configurações específicas de produtos.

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.

Prototype

Pseudocódigo

Neste exemplo, o padrão Prototype permite que você produza cópias exatas de objetos geométricos, sem acoplamento com o código das classes deles.



Clonando um conjunto de objetos que pertencem à uma hierarquia de classe.

Todas as classes de formas seguem a mesma interface, que fornece um método de clonagem. Uma subclasse pode chamar o método de clonagem superior antes de copiar seus próprios valores de campo para o objeto resultante.

```
// Protótipo base.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    // Um construtor normal.
    constructor Shape() is
        // ...

    // O construtor do protótipo. Um objeto novo é inicializado
    // com valores do objeto existente.
    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    // A operação de clonagem retorna uma das subclasses Shape.
    abstract method clone():Shape


// Protótipo concreto. O método de clonagem cria um novo objeto
// e passa ele ao construtor. Até o construtor terminar, ele tem
// uma referência ao clone fresco. Portanto, ninguém tem acesso
// ao clone parcialmente construído. Isso faz com que o clone
// resultante seja consistente.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        // Uma chamada para o construtor pai é necessária para
        // copiar campos privados definidos na classe pai.
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)


class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius
```

```
    method clone():Shape is
        return new Circle(this)


// Em algum lugar dentro do código cliente.
class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()
        shapes.add(anotherCircle)
        // A variável `anotherCircle` contém uma cópia exata do
        // objeto `circle`.

        Rectangle rectangle = new Rectangle()
        rectangle.width = 10
        rectangle.height = 20
        shapes.add(rectangle)

    method businessLogic() is
        // O protótipo arrasa porque permite que você produza
        // uma cópia de um objeto sem saber coisa alguma sobre
        // seu tipo.
        Array shapesCopy = new Array of Shapes.

        // Por exemplo, nós não sabemos os elementos exatos no
        // vetor shapes. Tudo que sabemos é que eles são todos
        // shapes. Mas graças ao polimorfismo, quando nós
        // chamamos o método `clone` em um shape, o programa
        // checa sua classe real e executa o método de clonagem
        // apropriado definido naquela classe. É por isso que
        // obtemos clones apropriados ao invés de um conjunto de
        // objetos Shape simples.
        foreach (s in shapes) do
            shapesCopy.add(s.clone())

        // O vetor `shapesCopy` contém cópias exatas dos filhos
        // do vetor `shape`.
```