2025-2 / Lorenzon : OO - Design Patterns - Builder

Design Patterns – Builder

O Builder é mais um dos padrões criacionais do catálogo GoF (Gang of Four). Ele é usado quando precisamos construir objetos complexos passo a passo, sem expor diretamente o processo de construção ao cliente. A ideia é separar a construção do objeto da sua representação final, permitindo criar diferentes representações de um mesmo objeto.

Seu uso é conveniente quando um objeto tem muitos atributos opcionais ou configuráveis. Quando o construtor tradicional (new) ficaria com parâmetros longos e confusos (o famoso telescoping constructor problem).

Quando queremos garantir imutabilidade e clareza no processo de criação. Depois que o objeto final é criado com .build(), ele não pode mais ser alterado. Isso é possível porque: Os atributos da classe são private e final. Não existem setters públicos para mudar o estado. Toda a configuração é feita somente dentro do Builder.

A construção de um exemplo também com aplicação de chamadas individuais e citando já um exemplo combinado de ferramentas:

Inner Class: Uma inner class é simplesmente uma classe declarada dentro do corpo de outra classe. Elas servem para organizar melhor o código quando uma classe só faz sentido existir dentro de outra. A classe Interna consegue acessar **atributos e métodos da classe Externa**, inclusive privados. São uma forma de **agrupar lógica relacionada** dentro de uma classe maior, e dão poder extra de **acessar membros privados** da classe externa. São utilizadas:

- Quando a classe tem sentido **apenas dentro de outra** (ex.: Builder dentro de Pessoa).
- Para **organizar** e evitar poluir o pacote com várias classes pequenas.
- Para **criar rapidamente implementações** de interfaces/abstract classes (anonymous inner class).
- Em padrões como **Builder**, **Observer**, **Adapter** etc.

As classes inners ainda podem ser locais de um método, não estáticas (precisando de um objeto da classe externa para serem acessadas, estáticas (como neste exemplo: Nested Static Class) e anônimas.

Fluent Interface : técnica de retornar this para permitir encadeamento de chamadas.

Method Chaining: nome mais técnico do encadeamento.

```
public class Pessoa {
    // Atributos
    private final String nome;
    private final int idade;
    private final String email;
    private final String telefone;
    private final String cidade;
```

```
    // Construtor privado
    private Pessoa(Builder builder) {
        this.nome = builder.nome;
        this.idade = builder.idade;
        this.email = builder.email;
        this.telefone = builder.telefone;
        this.cidade = builder.cidade;
    }
```

```
    // Classe interna estática Builder
    public static class Builder {
        private String nome;
        private int idade;
        private String email;
        private String telefone;
        private String cidade;

        public Builder setNome(String nome) {
            this.nome = nome;
            return this;
        }

        public Builder setIdade(int idade) {
            this.idade = idade;
            return this;
        }

        public Builder setEmail(String email) {
            this.email = email;
            return this;
        }
```

```
        public Builder setTelefone(String telefone) {
            this.telefone = telefone;
            return this;
        }

        public Builder setCidade(String cidade) {
            this.cidade = cidade;
            return this;
        }

        public Pessoa build() {
            return new Pessoa(this);
        }
    }
```

```
    @Override
    public String toString(){

        return "Nome: " + nome + "," + "\n" +
               "Idade: " + idade + "," + "\n" +
               "eMail: " + email + "," + "\n" +
               "Telefone: " + telefone + "," + "\n" +
               "Cidade: " + cidade + ".";

    }
}
```

```
public class App {

    public static void main(String[] args) {

        Pessoa.Builder builder = new Pessoa.Builder();

        builder.nome("Lorenzon");

        builder.idade(34);

        builder.email("vljunior@unochapeco.edu.br");

        builder.cidade("Chapecó-SC");

        Pessoa pessoa1 = builder.build();

        System.out.println(pessoa1);


        //usando técnica de Fluent Interface + Method Chaining

        Pessoa pessoa2 = new Pessoa.Builder()

                .setNome("João")

                .setIdade(30)

                .setEmail("joao@email.com")

                .setCidade("São Paulo-SP")

                .build();

        System.out.println(pessoa2);


        Pessoa pessoa3 = new Pessoa.Builder()

                .setNome("João")

                .setIdade(30)

                .build();

        System.out.println(pessoa3);

    }

}
```

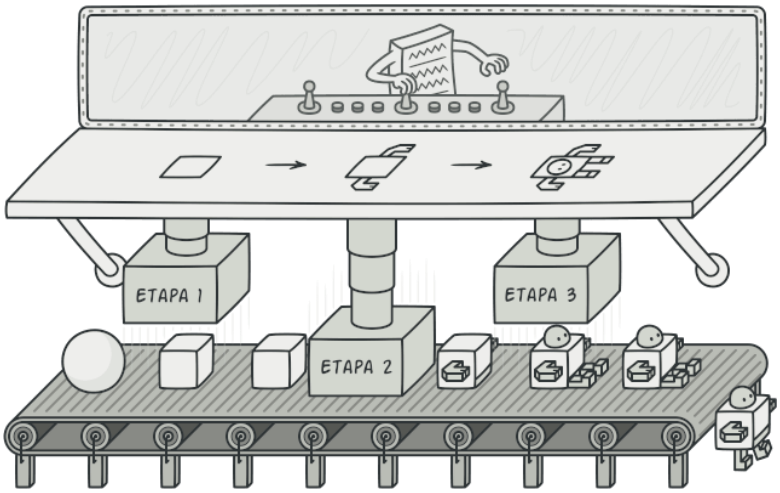
A classe Builder interna é a única forma de garantir que não se crie um objeto Pessoa pelo seu construtor padrão ou qualquer outro construtor, já que assim conseguimos privar o mesmo. Se a classe inner estivesse fora, o construtor seria público, quebrando a ideia do padrão.

Há conveniência de uso de código simplificado com Fluent Interface + Method Chaining, evitando-se linhas através do acoplamento que esta técnica possibilita usar.

A classe inner pode ser posicionada conforme desejado na classe mãe. Note que ela é pública, para que seja possível utilizar a chamada do seu construtor padrão.

Builder

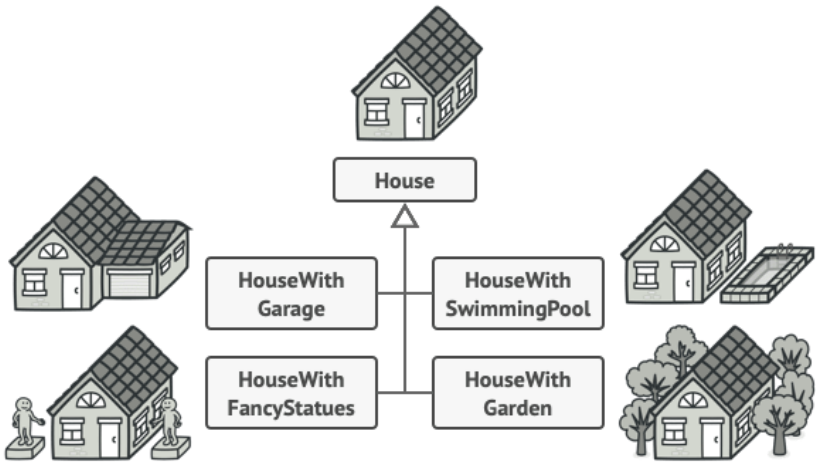
O **Builder** é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.



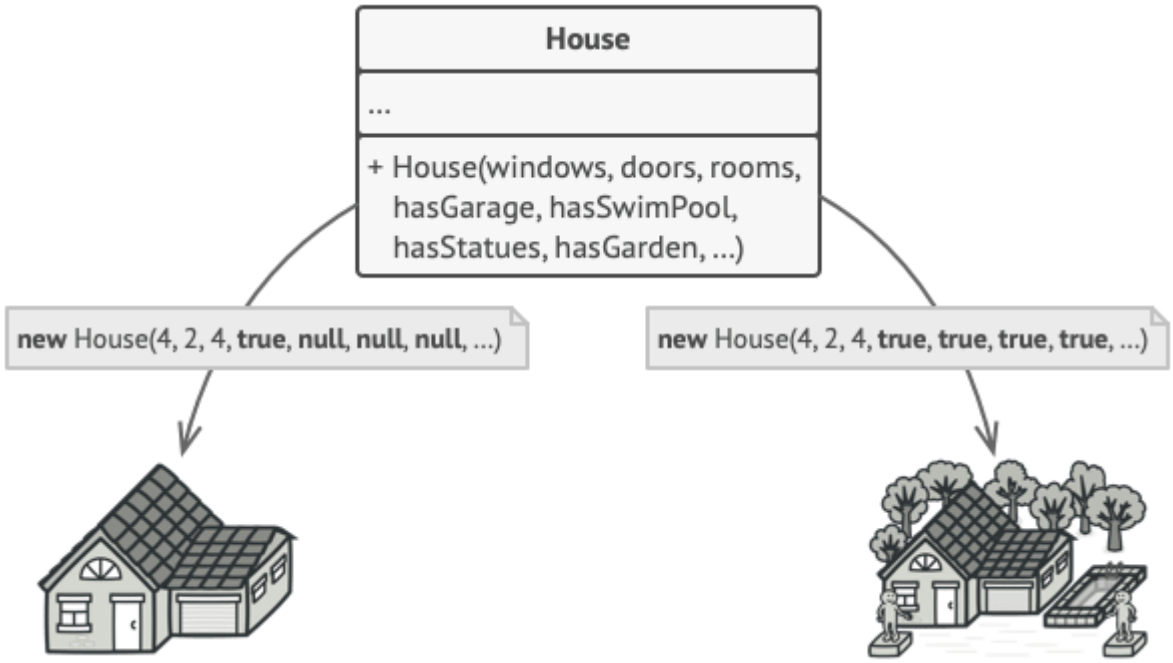
Por exemplo, vamos pensar sobre como criar um objeto Casa. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas, e construir um teto. Mas e se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como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



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.

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.

O Builder é amplamente usado nas bibliotecas principais do Java:

- **java.lang.StringBuilder#append()** (unsynchronized)
- **java.lang.StringBuffer#append()** (synchronized)
- **java.nio.ByteBuffer#put()** (também em **CharBuffer**, **ShortBuffer**, **IntBuffer**, **LongBuffer**, **FloatBuffer** e **DoubleBuffer**)
- **javax.swing.GroupLayout.Group#addComponent()**

- Todas as implementações de **java.lang.Appendable**