What is the Builder pattern, and what problem does it solve in software design?	Describe the structure of the Builder pattern. What are the main components involved?	What are the benefits of using the Builder pattern in software development? Provide some practical examples.
Builder é um padrão de projeto de software criacional que permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.	O padrão Builder está dividido em 4 aspectos: director — constrói um objeto utilizando a interface do builder; builder — especifica uma interface para um construtor de partes do objeto-produto; concrete builder — define uma implementação da interface builder, mantém a representação que cria e fornece interface para recuperação do produto; product — o objeto complexo acabado de construir. Inclui classes que definem as partes constituintes.	As vantagens do padrão Builder incluem: Permite variar a representação interna de um produto. Encapsula código para construção e representação. Fornece controle sobre as etapas do processo de construção. Podendo ser usado por exemplo em: Construção de Documentos HTML: Um Builder pode ser usado para construir documentos HTML de forma programática, permitindo a adição de elementos, atributos e conteúdo de maneira flexível.
The Builder pattern is a creational design pattern used in software design to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations. This pattern is particularly useful when an object needs to be constructed step by step, with a large number of optional components or configuration parameters, or when the construction process can vary based on the type of product being built.	The main components involved in the Builder pattern are: Product: The complex object being constructed. Builder Interface: Defines the steps to construct the product. ConcreteBuilder: Implements the builder interface and constructs the product. Director: Controls the construction process by calling the builder's methods in a specific order. Client: Uses the director to construct the product	The benefits of using the Builder pattern in software development include: Flexibility: It allows for the creation of different representations of an object using the same construction process. Readability: It makes the code more readable and maintainable by separating the construction logic from the business logic. Avoiding Telescoping Constructors: It helps avoid the need for constructors with multiple parameters, which can become unwieldy and error-prone. Immutable Objects: It supports the creation of immutable objects by constructing the object gradually before making it immutable. Configurable Object Creation: It provides a more flexible and readable way to specify configurations or variations of an object. Practical examples of using the Builder pattern include constructing complex objects like cars, where different configurations (e.g., sports car, SUV) can be built step by step using specific builders. Another example is constructing a pizza, where the user can choose various toppings and sizes, and the construction process is encapsulated in a builder, making the code cleaner and more manageable
Its used to simplify the construction of a very complex object. It solves the different representation of a complex object, making it easier to "build" due to only changing some parts and not the whole.	Abstraction of the builder, its implementations and the director, that states how the object is built.	It simplifies the creation of very complex objects. If we want to build a car, a very complex object, we should build each component first (engine, chassis, etc), and then assemble it in the end (using the director). This also allows to make changes to each component, allowing different representations

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

O padrão Builder resolve os seguintes problemas:
- Simplifica a criação de objetos complexos com muitos atributos.
- Evita construtores com muitos parâmetros e torna a criação de objetos mais legível.
- Permite a configuração flexível de objetos e a definição apenas dos atributos necessários.
- Facilita a adição de novos atributos ou versões de construção sem impactar a interface pública

O padrão Builder é usado para construir objetos complexos passo a passo, separando o processo de construção da representação final do objeto. Ele resolve problemas de complexidade na criação de objetos, especialmente quando há muitos parâmetros opcionais ou requisitos de inicialização complicados.

Os principais componentes envolvidos no padrão Builder são os seguintes:

1 - Builder: A interface do Builder define o processo de construção passo a passo para criar o objeto complexo. Ele declara um conjunto de métodos responsáveis pela construção de diferentes partes do objeto. Esses métodos normalmente retornam a própria instância do construtor para permitir o encadeamento de métodos.

2 - Construtores Concretos: As classes Concrete Builder implementam a interface Builder e fornecem a implementação específica para construir diferentes partes do objeto complexo. Cada construtor concreto mantém uma referência ao objeto que está sendo construído e implementa os métodos do construtor de acordo.

3 - Produto: O Produto representa o objeto complexo que está sendo construído. Normalmente contém diversas partes ou atributos que são construídos pelo construtor. A classe de produto não expõe suas etapas de construção nem a capacidade de modificar diretamente suas peças.

4 - Diretor: A classe Diretor é responsável por gerenciar o processo de construção. Ele recebe uma instância do construtor e fornece uma

Builder Interface: Define as operações necessárias para construir parte do produto final. Isso inclui métodos para construir diferentes partes do objeto complexo.

Concrete Builder: Implementa a interface do construtor e fornece a implementação específica para construir as diferentes partes do produto. Cada tipo de produto pode ter seu próprio.

Produto: Representa o objeto complexo sendo construído. Pode ser uma classe complexa com várias partes ou um simples objecto.

Cliente: É quem inicia o processo de construção, interagindo com o construtor diretamente para criar o objeto complexo.

- Podemos construir objetos passo a passo, adiar etapas de construção ou executar etapas recursivamente;
- Podemos reutilizar o mesmo código de construção ao construir diversas representações de produtos;
- Princípio da Responsabilidade Única - Podemos isolar códigos de construção complexos da lógica de negócios do produto.

O padrão Builder oferece várias vantagens no desenvolvimento de software. Ele separa a construção de objetos complexos de sua representação final, o que melhora a legibilidade e a manutenção do código. Além disso, fornece flexibilidade ao permitir a construção de diferentes variações de objetos usando o mesmo processo de construção. Isso é útil, por exemplo, ao lidar com objetos que possuem múltiplas configurações ou parâmetros opcionais. Por meio do encapsulamento da lógica de construção dentro da classe Builder, o padrão simplifica o código do cliente, tornando-o mais fácil de usar e reduzindo dependências.

Um exemplo concreto onde o padrão Builder pode ser aplicado é na produção de documentos, como relatórios ou faturas. Ao empregar um Builder, é possível construir a estrutura desses documentos de forma incremental, adicionando seções, parágrafos e opções de formatação à medida que necessário. Essa abordagem simplifica significativamente o processo de criação desses documentos, tornando-o mais dinâmico e flexível.

Helps to construct complex object step by step. It allows you to create objects of a class by providing a flexible solution for constructing the object with various configurations.
The main idea behind the Builder pattern is to separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

Enhances code readability, maintainability, and flexibility by providing a structured approach to constructing complex objects.

The builder pattern allows you to construct complex objects one step at a time, allowing the construction of different types and representations of an object with the same construction code. It solves a problem of, for example building a house with many differing parameters.

You might make the program too complex by creating a subclass for every possible configuration of an object
Builder suggest other approach.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Builder is a creational design pattern that creates complex objects step by step. This pattern allows to produce different types and representations of an object using the same construction code.
Without this pattern, initialization code is usually buried inside a monstrous constructor with lots of parameters, or scattered all over the client code.

Builder Interface/Abstract Class:
-This is an interface or abstract class that declares the methods for building individual parts of the complex object.
-It typically includes abstract methods for setting different attributes or configurations of the object.
-This interface provides a contract for all concrete builders to follow.

Concrete Builders:
-Concrete builder classes implement the Builder interface/abstract class.
-Each concrete builder provides a specific implementation for building the object.
-These classes are responsible for constructing different representations of the complex object.

Director (optional):
-The Director class is responsible for orchestrating the construction process using a builder object.
-It may provide a higher-level interface for constructing the object, coordinating the sequence of steps required for construction.
-The Director class is optional and can be omitted if the client code directly interacts with the builder objects.

Product:
-The Product is the complex object being

The builder contains a Director, a builder, and at least 2 concrete builders, allowing the construction of two different types of objects

The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

Having a director class in your program isn't strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

The Director execute a Builder interface that has all basic characteristics which then we have ConcreteBuilders to specify the particular prodcuts with specific characteristics.
Products are the resulting objects and the Client associate objects and directors.

1 - Builder
2 - ConcreteBuilder
3 - Product
4 - Diretor
5 - Client

Encapsulation of construction logic, flexibility in object creation, immutability and consistency, separation of concerns and code readability and maintainability.

You can construct objects step-by-step, defer construction steps or run steps recursively
You can reuse the same construction code when building various representations of products
You can isolate complex construction code from the business logic of the product, following the single responsibility principle

For example, let's think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.
The pattern organizes object construction into a set of steps (buildWalls, buildDoor, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

- Objects are constructed step-by-step, deferring construction steps or running steps recursively.
- Construction code can be reused when building various representations of products.
- Single Responsibility Principle. Complex construction code can be isolated from the business logic of the product.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
It allows us to have constructors with less parameters when creating a complex object For that, the Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.
To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

The Builder pattern consists on having a seperate class to create objects. It simplifies the creation of complex objects and enables the same class to create different representations of an object

The Builder Design Pattern is a creational pattern used in software design to construct a complex object step by step. It allows the construction of a product in a step-by-step fashion, where the construction process can vary based on the type of product being built. The pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
This pattern solves the problem of having a code with lots of subclasses, when implementing complex objects, which some aren't even used, making the constructor calls pretty ugly.

The Builder pattern is a creational design pattern that allows for the step-by-step creation of complex objects using the same construction code. This pattern is useful when an object needs to be constructed with many optional components or configurations, and you want to provide a clear separation between the construction process and the actual representation of the object.

One of the main problems with the Builder pattern is the boilerplate code required to implement it. This boilerplate code includes the creation of the builder interface, concrete builder classes, and the product class. Each time you need to create a new builder, you have to write a lot of repetitive code, which can make the codebase larger and harder to maintain.

The builder pattern has a director, that is connected to the interface that defines the methods to build each part of the "build". He defines what will be built, and he runs the different concrete builder classes. Each builder class has methods to build different components, a reset and a method to get the final result.

Director - the main class
Builder interface
Builders - Create the objects
Products - The objects created
Object

The main components involved are the Builder and the Director. The builder separates objects and create functions, organizing object construction into a set of steps. The Director is the element defines the order in which every step is implemented in order to achieve the desired goal and not a different one with the same components. In addition, the director class completely hides the details of product construction from the client code. It is important to highlight though that this class is not mandatory.
Main components:

Product: This is the final complex object that will be built. It defines the attributes and functionalities of the end result.
Builder Interface (or Abstract Builder): This interface defines the methods for constructing parts of the product. It acts as a blueprint for concrete builders.
Concrete Builder: This class implements the Builder interface and provides specific steps to build the product. There can be multiple concrete builders for creating different variations of the product.
Director (Optional): This component (not always used) takes the Builder interface as input and uses it to call the specific methods to build the product in a step-by-step manner. It acts like a conductor, orchestrating the construction process.

Structure:

The concrete builder classes implement the Builder interface, providing concrete implementations for building each part of the product.
The client interacts with a concrete builder, calling its methods to configure different aspects of the product.
The builder methods typically return the builder object itself, allowing for method chaining

With this pattern we can construct objects step-by-step, defer construction steps or run steps recursively, reuse the same construction code when building various representations of products and we comply with the Single Responsibility Principle by isolating complex construction code from the business logic of the product.
A practical example would be configuring a food order, for example a burguer, we have to decide on sauce, ingredients and bread, a BurguerBuilder can be used to define the burguer step-by-step. Customers can choose everything using the methods provided by the builder, this allows for a clear and flexible burguer configuration.

The builder pattern separates the object from its construction and representation, which makes it possible to change them seperatly

You can construct objects step-by-step, defer construction steps or run steps recursively.
You can reuse the same construction code when building various representations of products.
Single Responsibility Principle. You can isolate complex construction code from the business logic of the product

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

A practical example is for example when you need to build custom computers with varying CPUs, RAM, and storage options. The Builder pattern can be used to construct these computers through a step-by-step process, allowing for flexibility in creating computers with different configuration.

This example demonstrates how the Builder pattern can be used to construct a complex object (a computer) step by step, allowing for the creation of different configurations based on user preferences. The Computer class is the product, Builder is the interface, GamingComputerBuilder is the concrete builder, ComputerDirector is the director, and the Client assembles the product using the builder and director. The Builder pattern is particularly useful in scenarios where an object requires a large number of optional components or configurations, or when the construction process needs to be clearly separated from the representation of the object.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. The problem it solves is imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders. The main components are the Mainbuilder who has access to all builders in a specific order and the director

You can construct objects step-by-step, defer construction steps or run steps recursively.
You can reuse the same construction code when building various representations of products.
Single Responsibility Principle. You can isolate complex construction code from the business logic of the product. For example, let's think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof.

É um padrão de design Creational que permite construir objetos complexos passo a passo. O padrão oferece uma maneira de criar um objeto, separando a lógica da construção da representação do objeto, permitindo que o mesmo processo de construção crie diferentes representações de objetos.

O principal problema que o padrão Builder resolve é a construção de objetos que requerem múltiplas etapas ou parâmetros.

O padrão Builder aborda esses problemas fornecendo uma separação clara entre o processo de construção e a representação do objeto. Existe uma classe de construtor responsável por construir o objeto. A classe do construtor contém métodos para definir as propriedades ou os parâmetros do objeto, permitindo que o código do cliente especifique apenas os atributos desejados. A classe do construtor, então, lida com a lógica de construção real com base nos parâmetros fornecidos.

A interface do Builder declara etapas de construção do produto que são comuns a todos os tipos de construtores.
Concrete Builders oferece diferentes implementações das etapas de construção, podem produzir produtos que não seguem a interface comum.
Products são os objetos resultantes, construídos por diferentes builders que não precisam pertencer à mesma interface ou hierarquia da classes.
A classe Director define a ordem pela qual as etapas de construção são chamadas, logo permite criar e reutilizar configurações específica de produtos.
O Client tem a opção de associar um objeto builder ao Director de duas formas. A primeira e mais comum é passar o objeto builder como parâmetro no construtor do Director. Dessa forma, o Director usará o mesmo objeto builder para todas as construções futuras. No entanto, existe uma abordagem alternativa em que o cliente passa o objeto builder diretamente para o método de produção do Director. Nesse caso, é possível utilizar um builder diferente a cada vez que o Director realizar uma construção.

O padrão Builder separa o processo de construção de um objeto complexo da sua representação final. Isso permite que o código de construção seja independente da estrutura interna do objeto, facilitando a criação de objetos complexos passo a passo, adicionando gradualmente diferentes componentes ao objeto final. Assim, permite um melhor controlo da criação do objeto e uma mais fácil reutilização do código, uma vez que torna-o mais legível.
Benefícios:

Separar a construção de um objeto complexo da sua representação, de forma a que
a construção (passo a passo) possa criar diferentes representações.

Isto advém do problema de na construção de um objeto complexo geralmente implica um construtor com imensos parâmetros, alguns dos quais nem sempre necessitamos. Uma solução seria ter uma classe base com os parâmetros principais e as variações tomarem forma em subclasses. No entanto, são tantas que iriamos exponenciar a complexidade.

Assim, ao retirar a criação do objeto da sua classe e movê-la para outra, vamos fazer um builder, um elemento secundário onde os vários elementos são armazenados para depois serem representados no primário (o objeto).

Devemos assim criar um abstract builder que é responsável por fazer os sets e um Director, responsável por gerir a construção do objeto.

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

Os Concrete Builders fornecem diferentes implementações das etapas de construção. Os construtores destas podem produzir produtos que não seguem a interface comum.

Os produtos são objetos resultantes. Os produtos construídos por construtores diferentes não precisam pertencer à mesma hierarquia de classes ou interface.

A classe Director define a ordem de chamada das etapas de construção, para que você possa criar e reutilizar configurações específicas de produtos.

O Cliente deve associar um dos objetos construtores ao diretor. Normalmente isso é feito apenas uma vez, através dos parâmetros do construtor do diretor. Em seguida, o diretor usa esse objeto construtor para todas as construções futuras. Porém, existe uma abordagem alternativa para quando o cliente passa o objeto construtor para o método de produção do diretor. Nesse caso, você pode usar um construtor diferente cada vez que produzir algo com o diretor.

Separação da construção e representação: Permite construir objetos complexos passo a passo, separando o processo de construção da representação final do objeto. Isso facilita a produção de diferentes representações ou versões de um objeto usando o mesmo processo de construção.
Encapsulamento: O cliente não precisa conhecer os detalhes internos da construção do objeto. Isso simplifica a interface de criação de objetos complexos e reduz a dependência entre o código cliente e as classes que formam o objeto complexo.
Controle sobre o processo de construção: Permite ajustar o objeto final em tempo de execução, baseando-se em parâmetros ou configurações dinâmicas.

Exemplos práticos:

Montagem de Pizzas em uma Aplicação de Delivery: Um builder pode ser utilizado para permitir aos clientes montar suas pizzas passo a passo, escolhendo a massa, o molho, os ingredientes, e outros adicionais. Cada escolha é um passo na construção do objeto final, que é a pizza personalizada.

Configuração de Computadores em Lojas Online: Ao comprar um computador online, clientes podem personalizá-lo de acordo com suas necessidades, escolhendo diferentes componentes como processador, memória RAM, placa gráfica, e armazenamento. Um builder facilita este processo ao permitir a seleção e

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

O objeto que se pretende instanciar. O Builder que tem os componentes da criação do objeto, sendo o construtor uma chamada genérica e depois efetuados comandos no builder para construir a instancia desejada.

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

O design pattern Builder permite construir objetos complexos através de um processo passo a passo, do estilo linha de montagem. Desta forma, é possível criar diferentes tipos de um objeto através da seleção de diferentes componentes.

The main components involved in the Builder Pattern are the following:
Director: Orchestrates the object creation by consuming the Builder interface. It specifies the sequence of construction steps.
Builder Interface: Outlines the blueprint for the construction steps. It typically includes methods for constructing individual parts of the product.
Concrete Builder: Implements the Builder interface, providing specifics for each construction step and managing the product's assembly.
Product: The intricate object that's being constructed. Notably, the system usually remains unaware of the Concrete Builder's presence, interfacing only with the Director and Product.

Os benefícios do Builder Pattern surgem da necessidade da criação de objetos complexos com um grande número de componentes ou parâmetros de configuração, este pattern é muito útil quando um objeto precisa de ser construído passo a passo. Alguns dos cenários práticos do uso deste Pattern são, por exemplo, queremos providenciar uma interface comum para a construção de diferentes representações de um objeto, quando se quer criar objetos imutáveis, e o pattern permite as suas construções graduais antes de o fazer assim, etc.

Este padrão consiste em construir objetos complexos passo a passo, sendo que também podemos criar objetos de diferentes tipos e representações através dos mesmo código de construção

A interface Builder, a class Director e os concrete builders

Construir objetos passo a passo, reusar o mesmo código de construção e aplicar o SRP
Todas as implementações de java.lang.Appendable

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.
- The Builder interface declares product construction steps that are common to all types of builders.
- Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don’t follow the common interface.
- Products are resulting objects. Products constructed by different builders don’t have to belong to the same class hierarchy or interface.
- There may be a Director class that defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- The Client must associate one of the builder objects with the director. Usually, it’s done just once, via parameters of the director’s constructor. Then the director uses that builder object for all further construction.

The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. A builder doesn’t expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.
- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

Builder is a creational design pattern that lets you construct complex objects step by step. It is allows you to produce different types and representations of an object using the same construction code.

Product is the complex object that the Builder pattern is responsible for constructing;
Builder is an interface or an abstract class that declares the construction steps for building a complex object;
ConcreteBuilder classes implement the Builder interface, providing specific implementations for building each part of the product;
Director is responsible for managing the construction process of the complex object;
Client is the code that initiates the construction of the complex object.

Allows you to vary a product's internal representation;
Encapsulates code for construction and representation;
Provides control over the steps of the construction process.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

The Builder interface declares product construction steps that are common to all types of builders.

Concrete Builders provide different implementations of the construction steps.

Products are resulting objects.

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders. It lets you construct complex objects step by step. The Builder doesn’t allow other objects to access the product while it’s being built.

The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

The Client must associate one of the builder objects with the director.

You can construct objects step-by-step, defer construction steps or run steps recursively.
You can reuse the same construction code when building various representations of products.
You can isolate complex construction code from the business logic of the product.

The Builder pattern is a creational design pattern used to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

This pattern is particularly useful when dealing with objects that have many configurable parameters or optional features.

The Builder pattern solves several problems in software design: Complex object creation, Flexible object creation, Encapsulation of construction logic and more.

O padrão Builder permite a produção de diferentes tipos e representações de objetos a partir do mesmo código de construção.
Resolve o problema de construtores muito grandes e complexos para objetos complexos.

Builder é um padrão de design criacional que permite construir objetos complexos passo a passo. O padrão permite produzir diferentes tipos e representações de um objeto utilizando o mesmo código de construção.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
It solves the problem of initializing a complex object that requires laborious, step-by-step initialization of many fields and nested objects.
To solve the problem, the Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

It's structure typically involves the following main components:
1.Builder Interface;
2.Concrete Builders;
3.Director;
4.Product;
5.Client
This structure allows for the construction of complex objects with varying configurations and promotes separation of concerns and flexibility in object creation.

A estrutura deste padrão:
- Interface Builder -> Declara etapas de construção
- Builder Concretos -> Fornece diferentes implementações das etapas de construção
- Objetos Resultantes -> Produto resultante dos Builders
- Director -> Define a ordem da chamada das etapas
- Client -> Faz a associação entre os objetos e o Director.

A interface do Builder declara etapas de construção do objeto que é comum a todos os outros tipos de objetos (ex. construção). Os Concrete Builders fornecem diferentes implementações das etapas de construção. Os Concrete Builders podem produzir ojetos que não seguem a interface comum.

The Product represents the complex object that the Builder pattern constructs.
The Builder is an interface or an abstract class that declares the construction steps for building the complex object.
ConcreteBuilder classes implement the Builder interface, it provide specific implementations for building each part of the product. Each ConcreteBuilder is tailored to create a specific variation of the product.
The Director is responsible for managing the construction process of the complex object.
The Client initiates the construction of the complex object.

The benefits of using the Builder pattern are:

Separation of concerns: Construction process is separated from object representation.
Simplified object creation: Step-by-step approach makes object construction easier to manage.
Flexibility in object construction: Different object representations can be created by varying construction process.
Encapsulation of construction logic: Complexity of object creation is abstracted within builder classes.
Reusability: Builders can be reused for creating multiple instances, reducing code duplication.
Practical Example:
Document Builder: In a document processing application, a builder can be used to construct documents with different structures, formatting styles, and content elements. For example, a document builder could construct a text document with headings, paragraphs, and images, or a spreadsheet document with rows, columns, and formulas.

Os benefícios são:
- Construção de objetos passo a passo
- Reutilização do código de construção
- Permite seguir o primeiro principio do SOLID.

Exemplo Prático:
Imaginemos a construção de uma casa, que envolve vários passos e diferentes níveis de complexidade. Para além disso existem também casas com jardim, com piscina, com quintais, o padrão Builder permitiria simplificar estes objetos complexos noutros mais simples de instanciar.

Construir objetos passo a passo, adiar etapas de construção ou executar etapas recursivamente.
Reutilizar o mesmo código de construção ao construir diversas representações de produtos.
Princípio da Responsabilidade Única. Poder isolar códigos de construção complexos da lógica de negócios do produto.
The Builder pattern separates the construction and representation of an object.
It hides the internal details of object creation from the client code.
This separation enhances code readability and maintainability.
Flexible Object Construction:
Builders allow creating different types and representations of complex objects using the same creation code.
You can customize the construction process step by step, adapting to various scenarios.
This flexibility is especially useful when dealing with objects with multiple components or configurations.
Avoids Telescoping Constructor Pattern:
The Telescoping Constructor Pattern occurs when constructors have multiple parameters with different combinations.
Builders eliminate this issue by providing a clean way to construct objects without a long list of constructor arguments.
Improved Readability and Maintainability:
By encapsulating the construction logic within the builder, the client code remains concise.
Changes to the object's construction process are localized to the builder, making maintenance easier

The Builder pattern allows a step-by-step approach:

Size: "I'd like a large pizza."
Crust: "Thin crust please."
Toppings: "Can I add pepperoni?"

The builder Design Pattern is a creational pattern to construct a complex object step by step, where the construction process vary based on the type of product built
The pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Product, Builder, ConcreteBuilder, Director, Client

Construction of objects step by step, reusability of the same construction code when building various representations of products, isolating construction code from business logic of the product

---

A interface do Builder declara etapas de construção do produto que pertencem a todo o tipo de construtores.

O padrão Builder permite construir um objeto passo a passo, isolando o código de construção do código de representação. Isso torna o código mais simples de alterar, fácil de ler e manter.

O padrão Builder é um padrão de design criacional que tem como objetivo encontrar uma solução para o problema da construção de objetos complexos.
O principal problema que o Builder visa resolver é a necessidade de criar um objeto que possa ter várias representações diferentes ou que necessite de vários passos para a sua construção, que podem precisar de diversos parâmetros.

Este padrão é especialmente útil em situações onde um objeto precisa de ser criado e este possui muitos atributos, tornando a construção diretamente no construtor problemática devido à complexidade ou à legibilidade do código.

Sem o padrão Builder, o código pode acabar com muitos construtores, cada um lidando com diferentes combinações de parâmetros, ou com um único construtor com muitos parâmetros, o que pode ser confuso e levar a erros.

Os Concrete Builders fornecem implementações distintas das etapas de construção.Estes podem produzir produtos que não seguem a interface comum.
Os produtos que são construídos por construtores diferentes não precisam de pertencer à mesma hierarquia ou interface.

A classe Director define a ordem de chamada das etapas de construção, para que que seja possivel criar e reutilizar configurações específicas de produtos.

O Cliente deve associar um dos objetos construtores ao diretor. Normalmente isso é feito apenas uma vez, através dos parâmetros do construtor do diretor. O diretor irá usar esse objeto construtor para todas as construções futuras. Existe uma abordagem alternativa para quando o cliente passa o objeto construtor para o método de produção do diretor. Nesse caso, é possivel usar um construtor diferente cada vez que for produzido algo com o diretor.

Ao utilizar o padrão Builder, o processo de construção do objeto fica encapsulado no próprio builder. Isso significa que o objeto pode impor invariâncias internas sem expor a lógica de construção ao código cliente.

Este padrão é particularmente útil quando um objeto precisa ser construído em várias etapas ou requer uma configuração complexa.

Após o objeto ter sido construído, ele pode ser imutável e não necessitar de mudanças. Isto é vantajoso em ambientes multi-thread, onde a imutabilidade reduz erros e complexidade.

Com diferentes builders, podemos construir variações do mesmo objeto sem alterar o código que faz a construção. Isso permite uma maior flexibilidade e reutilização de código.

Adicionar novos campos ao objeto que está sendo construído não exige mudanças nos construtores existentes, apenas no builder. Isso mantém o código cliente limpo e livre das mudanças internas do objeto.

A interface Builder declara etapas de construção do produto que são comuns a todos os tipos de builders, os concrete Builders provém diferentes implementações das etapas de construção e os Produtos são os objetos resultantes. Para alem disso, existe a classe Diretor que define a ordem na qual as etapas de construção são chamadas, então pode-se criar e reutilizar configurações específicas de produtos.

O builder permite-nos criar objetos passo a passo, reutilizar o codigo de construção e obedece o principio de responsabilidade unica.

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.

Exemplo prático:
Uma concessionária de carros pode oferecer um sistema onde os clientes podem personalizar os seus veículos. O Builder permite selecionar motor, cores, rodas, acessórios etc., construindo o carro de acordo com as escolhas do cliente.

The builder design pattern allows us to deconstruct the building of complex objects into smaller and easier steps. If we needed a new object that's similar but slightly different from the others we would only need to add or modify steps of the builder to create this new object, simplyfying our work.

The builder pattern involves a builder (interface or abstract class) that dictates what methods to use to build the object, a concrete builder, that implements the builder interface and builds the desired product and a director, that provides a high level interface to help guide the building of the product.

The builder pattern allows us to make changes to our codebase (namely in creating new objects) easily, as it is quite clear where the changes have to be made to create a new object. It also allows the user to easily create objects without knowing the exact methods to do so.

	Builder, Concrete builders, Products, Director, Client. The Builder interface declares product construction steps that are common to all types of builders. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface. Products are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface. The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products. The Client must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.	
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.		You can construct objects step-by-step, defer construction steps or run steps recursively. You can reuse the same construction code when building various representations of products. Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. This pattern solves the problems of creating multiple and excessive amounts of subclasses and the problem of creating a general constructor with an enourmous amount of parameters.	There are building blocks/functions: the builder, that provides the implementation of the construction steps. Besides that, there is the Director (that gives the construction order of the building steps) and the Client, which is the program that call upon the directors.	You can construct objects step-by-step, defer construction steps or run steps recursively. You can reuse the same construction code when building various representations of products. Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.
O Builder é um padrão que permite construir objetos complexos passo a passo. O padrão permite produzir diferentes tipos e representações de um objeto usando o mesmo código de construção, ou seja, separa a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar diferentes representações. Analisa uma representação complexa e crie uma entre várias alvos.	Ele separa a construção de um objeto complexo de sua representação, permitindo a criação do objeto passo a passo. Isso permite a construção de objetos com diferentes características usando o mesmo processo de construção. Utiliza interfaces Builder em que declaram todos os tipos comuns de um produto. Também define o classes para que possam ser reutilizadas em configuraçoes especificas.	Em um sistema de construção de carros. O Builder pode ser usado para construir diferentes tipos de carros com as mesmas etapas de construção, mas com características específicas definidas durante o processo de construção.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

This design pattern avoids the creation of ugly constructurs where most of the parameters are unused

É um creational pattern que te permite construir objetos complexos, numa lógica passo-a-passo, que permita ao cliente configurar o objeto que quer que seja criado.

Structure:

- The Builder: an interface that declares product construction steps that are common to all types of builders;
- The Concrete Builders: classes that provide different implementations of the construction steps. Concrete Builders may produce products that don't follow the common interface.
- The Products: are the resulting objects. Those constructed by different builders don't have to belong to the same class hierarchy or interface.
- The Director: class that defines the order in which to call construction steps, so the client can create and reuse specific configurations of products.
- The Client: must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

O builder é uma classe segregada do objeto que se pretende instanciar. Este deve conter um método que, inicialmente, instancia o objeto e, para além deste, diversos outros que deverão permitir ao cliente decidir como deseja configurar o objeto principal.

Benefits:

- It is possible to construct objects step-by-step, defer construction steps or run steps recursively.
- It is possible to reuse the same construction code when building various representations of products.
- It is possible to isolate complex construction code from the business logic of the product.

Permite, de forma simples e eficiente, trabalhar com objetos de elevada complexidade. Dá mais liberdade ao cliente na configuração do objeto.
Por exemplo, a construção de uma casa implica várias etapas complexas que implicam que sejam feitas escolhas relativamente à construção da mesma.
O padrão Builder é um padrão de projeto de software usado para construir um objeto complexo passo a passo.

Com isso torna o código mais legível, manutenível e evita a necessidade de ter múltiplos construtores com diferentes combinações de parâmetros.

Os sues principais benefícios são:

Separação da Construção e Representação que consiste na construção de um objeto complexo passo a passo, ao mesmo tempo que permite a diferentes representações.

Encapsulamento isto é, não necessitamos conhecer toda a composição interna, pois usa uma interface para a criação de objetos complexos

Existe também controle fino sobre o processo de construção, isto é, o builder da controle completo sobre o processo de construção, incluindo a ordem em que as partes são montadas

A Imutabilidade de objeto final, isto é, um objeto construído ele pode ser imutável sem a necessidade de tornar todas as classes de campo imutáveis,

Exemplo: Sistema para uma lanchonete de montar Hambúrguer

The builder pattern is a creational design pattern in software engineering that allows for the step-by-step construction of complex objects using a builder object. The builder object separates the construction of a complex object from its representation, allowing for a more flexible and readable way to create these objects.

The builder pattern solves the problem of creating complex objects with multiple required and optional attributes, where calling the constructor with all the required parameters can become verbose and error-prone. By using a builder object, the code for creating these objects becomes more readable and maintainable, as the required and optional attributes can be set in a more fluent and expressive way.

O objetivo do builder pattern é construir objetos passo a passo possibilitando a geração de objetos diferentes com o mesmo código de construção. Resolvendo o problema de ao ter varios objetos com pequenas diferenças nao temos de criar uma fabrica com muitos parametros.

The stucture of the Builder Pattern is divided in several coponents. The first one is The Builder Interface, which declares product construction steps which are common to all types of builders. Next we have Concrete Builders that provide different implementations of the construction steps, which may or may not produce products that don't follow the common interface. The resulting objects are called Objects, these don't have to belong to the same class hierarchy or interface. The optional Director class defines the order in which the steps are to be executed, so it is possible to create and reuse specific configurations. Finally, there is the Client which must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction.

O Builder vai ter uma interface que declara as estapas de construção do produto que são comuns a todos os tipos de contrutor, as classes Concrete Builder que fornecem diferentes implementações das etapas de construção resultando produtos. A classe director que define a ordem de chamada das etapas de construção.

Temos várias formas de criar os objetos, podendo ser passo a passo ou recursivamente. Podemos reutilizar o mesmo codigo de construção para criar varias representaçoes do produto, alem de que segue o Single Responsibility Principle. Por exemplo ao contruir carros diferentes podemos usar o mesmo builder para os varios parametros do carro.

It is a creational design pattern that addresses the issue of constructing complex objects step-by-step in a clear and readable way. It separates the object construction process from its representation, allowing more control and flexibility.
It solves the problems of Complex Object Construction, Immutable Objects and Varied Object Configurations.

The Builder pattern's main components are Builder (interface that defines the methods for constructing parts of the product), Concrete Builder (concrete implementation of the Builder interface), Product (final object that is built), Director (defines order in which to call construction steps) and Client (associates one of the builder objects with the director).

The benefits are:
- It is possible to construct objects step-by-step, defering construction steps or run steps recursively;
- Reusability of the same construction code when building various representations of the product;
- Uses the SIngle Responsability Principle

A practical example would be an HTML Report Generator, an HTML Builder that can be used to define the report structure step-by-step, adding headers, paragraphs, tables and more, allowing customization of report formats.

Builder is a creational design pattern that allows the construction of complex objects step by step, this way it allows the production of different types and representations of an object using the same construction code.

The main components of the Builder pattern are:
-A builder interface that declares product construction steps common to all types of builders
-Concrete builders which provide different implementations to the construction steps, it can originate products that don't follow the common interface.
-Products, knowing that, if constructed by different builders, they may belong to a different class hierarchy or interface.
-The director class, which is a class that defines the order in which the construction steps are called.
-The client, it associates the objects of the builder with the director, via parameters of the directors constructor.

Some benefits of this pattern are:
-It follow the Single responsability principle
-It allows us to reuse the same construction code when building different representations of products
-It also allows to construct objects step by step, defer construction steps or run steps recursively
A good practical example of this Builder pattern is the String Builder of java. A more precise example is, for example, the building of a custom computer with specific components like CPU, GPU, RAM, storage, and peripherals. Using the Builder pattern can help in constructing the computer object step by step, allowing flexibility in choosing components and configurations.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Consider a scenario where the initialization of a complex object demands meticulous attention to detail, involving the configuration of numerous fields and nested components. Typically, such initialization logic is encapsulated within an unwieldy constructor method, laden with a multitude of parameters, or, in a more chaotic scenario, scattered throughout the client code.

For instance, let us contemplate the instantiation of a House object. Constructing a basic house entails erecting four walls, laying a floor, installing a door, fitting windows, and constructing a roof. However, should one aspire for a larger, more elaborate dwelling complete with amenities such as a backyard, heating system, plumbing, and electrical wiring, the complexity escalates.

One ostensibly straightforward solution involves extending the base House class and generating a series of subclasses to accommodate various parameter

1- The Builder interface declares product construction steps that are common to all types of builders.
2- Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don’t follow the common interface.
3- Products are resulting objects. Products constructed by different builders don’t have to belong to the same class hierarchy or interface.
4 - The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
5- The Client must associate one of the builder objects with the director. Usually, it’s done just once, via parameters of the director’s constructor. Then the director uses that builder object for all further construction. However, there’s an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

Some benefits would be:
- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

Practical examples:
When building a car, there are many configurable options such as the type of engine, transmission, interior upholstery, and optional features like sunroof or navigation system. Using the Builder pattern, you can construct cars step-by-step, allowing the user to specify each configuration option independently and in any order they prefer. For example, you can first specify the engine type, then the transmission, followed by interior options, and so on.

A estrutura do padrão Builder envolve quatro componentes principais:

1- Builder: É uma interface ou classe abstrata que define os métodos para construir as partes do objeto. Esses métodos representam as etapas ou partes do objeto que estão sendo construídas.
2- ConcreteBuilder: São classes concretas que implementam a interface ou herdam da classe abstrata Builder. Cada ConcreteBuilder define como construir uma parte específica do objeto.
3- Director: É uma classe que usa o Builder para construir o objeto passo a passo. O Director não precisa saber os detalhes de como o objeto é construído, apenas que ele precisa ser construído. O Director chama os métodos do Builder na ordem correta para construir o objeto.
4- Product: É o objeto complexo que está sendo construído. O Product é o resultado final da construção realizada pelo Builder.

O padrão Builder é um padrão de projeto de software que permite construir objetos complexos passo a passo. Ele é usado quando a construção de um objeto requer muitos passos ou quando esses passos precisam ser realizados de maneira específica. O padrão Builder separa a construção de um objeto da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.
Sem o padrão Builder, você teria que criar vários construtores com diferentes parâmetros ou métodos setter para configurar o objeto de várias maneiras. Isso pode levar a um código confuso e difícil de manter.

Alguns dos benficíos deste padrao sao:
1- Flexibilidade: Permite construir objetos de várias maneiras diferentes, facilitando a criação de objetos complexos com configurações variadas.
2- Legibilidade: Torna o código mais legível e fácil de entender, pois separa a construção de um objeto de sua representação.
3- Manutenção: Facilita a manutenção do código, pois adicionar novas formas de construção de um objeto não requer mudanças na classe do objeto.
4- Reutilização: O Builder pode ser reutilizado para construir diferentes tipos de objetos, desde que eles compartilhem uma estrutura de construção semelhante.

O padrão Builder é uma solução poderosa para construir objetos complexos de maneira flexível e legível.

The builder pattern consists of a new class called the builder, this class can either hold an instance of the final object or its constituent parts, then it provides methods relevant to the construction of the object, for example adding an element to an internal list, changing an attribute. Afterwards a final method is provided normally called build or getResult that will return the final object, this can be the object that was mutated or the result of the object instantiation from the parts that were defined.

This idea can be further extended by using an abstract builder that specifies the build steps that can then be specialized by concrete builders depending on the program needs.

The Builder pattern allows for the creation of complex objects step by step, allowing for the creation of different variants of the same class using the same construction code. An example can be the construction of a class with a lot of attributes and fields. Initialising them all with the constructor would make it so the constructor has to take a lot of parameters. The solution is to instantiate a new object with all the attributes set to null and then, step by step, complete the information.

The benefits of using a builder pattern are that it removes the necessity of creating a new class for every new parameter the superclass wants to add. A solution to that problem is to move those parameters into one class but that leads to huge ugly constructors which are also something we are trying to avoid. To solve this we use a builder whose purpose is to build an instance of a class step by step, this way we avoid making huge constructors where most of the fields are unused and we have a cleaner construction of the class by calling only the parameters we need.
A practical example of this is in a calendar class that has tons of parameters and tiny differences from one type of calendar to another. A builder helps to create the specific type of calendar I need by choosing the parameters I need step by step.

Builder: interface ou classe abstrata usada para definir as etapas necessárias para construir um objeto.
ConcreteBuilders: implementa a interface builder e fornece implementaçoes específicas para as etapas de contrução
Director: é opcional e responsável por gerir o processo de contrução com base num conjunto de etapas.
Products: apresenta o objeto complexo que está a ser construido.

É um padrão creacional que permite construir objetos complexos passo a passo.
Permite produzir diferentes tipos e representações de um objeto usando o mesmo construction code.
REduz a complexidade dos construtores, facilita a construção de objetos com muitos atributos e promove a reutilização de código de construção.

É possível construir obejtos passo a passo, adiar passos da construção e, até mesmo, correr passos recursivamente. Permite reutilizar o mesmo código de construção para várias representações dos produtos.
Single Responsibility Principle.

Existe uma classe Builder com atributos e setters para esses atributos, tendo ainda um método Build que retorna uma instância da classe alvo passando ao seu construtor os parâmetros da instância da classe Builder, ou então, a classe Builder tem como atributo uma instância da classe alvo, e nos seus setters vai usar os setters da classe alvo de modo a ir construindo a instância passo a passo, finalmente no método build retorna essa instância.

É um padrão criacional que permite a criação de objetos complexos, passo a passo.
Permite produzir diferentes tipos de representação de objetos usando o mesmo código de construção.
Resolve o problema de classes que têm um construtor com muitos parâmetros, simplificando o seu uso.

Permite construir objetos passo a passo, adiando etapas de construção ou executando etapas recursivamente.
Permite reutilizar o mesmo código de construção ao construir diversas representações de produtos, isto é, instâncias da mesma classe com diferentes características.
Respeita o Princípio da Single Responsability.
Por exemplo, o StringBuilder().
Por exemplo um carro tem vários componentes (motor, bancos, portas etc), por isso o seu construtor iria ter muitos parâmetros, podemos usar então um CarBuilder que têm métodos (setMotor, setBancos etc) que vai adicionando esses componentes à instância de Car, no método build retorna essa instância já com todos os atributos.

O builder pattern pode ter a seguinte estrutura um construtor que declara a construção do produto, construtores concreto onde fornece diferentes implementações das etapas de construção, produto que são objetos resultantes, diretor que define a ordem em que char as etapas de construção, e por fim o cliente que deve ser associado a um dos objetos do construtor e a componente main desse builder e o construtor porque sem ele nao se pode instanciar os objetos.

e um padrão de design que permite a construir objetos complexo passo a passo que permite produzir diferentes tipos e representações de um objeto usando o mesmo código.
a solução que ele resolve e de o código de construção da sua própria classe separando os construtores.

O padrão Builder é particularmente útil quando um objeto precisa ser construído passo a passo, com muitos componentes opcionais ou configurações, Ele permite separar o processo de construção da representação real do objeto, facilitando a criação de objetos complexos.

É um padrão que permite dividir a construção de objetos complexos em passos. Resolve o problema de construtores muito complexos permitindo a criação de diferentes tipos utilizando o mesmo codigo do construtor	- Interface builder que vai ter todos os tipos de builder - Builders concretos que têm as implementações da interface builder e retornam um produto - Os produtos são o resultado dos builders e não têm de ser todos do mesmo tipo - O Director define a ordem pela qual os métodos de construção são chamado e permite reutilizar configurações especificas - O Client é o que indica ao Director qual o Concrete Builder a utilizar	- Dá para construir step-by-step - Permite a reutilização - Segue o Single Responsible Principle
É um padrão que divide o construtor em vários passos, mantendo uma interface comum. Um problema que resolve é como instanciar diferentes subclasses do mesmo objeto. Impede também de chamar argumentos "null", chamando apenas os métodos necessários.	-Interface Builder, com os métodos comuns a todos os builders; -Concrete Builders,construtores que implementam todos os métodos do builder, mas também métodos mais específicos; -Produtos, são o resultado das chamadas dos builders, -Director, define a ordem em que se chamam oa passos dos builders,	Podem ser construidos objetos passo-a-passo, mudando a construção ou até chamado recurssivamente, Pode-se reutilizar o mesmo contrutor criando diferentes representações do mesmo produto, Ajuda a manter o Single-Responsibilty-Principle
O Builder é um padrão de design que permite construir objetos complexos passo a passo. O padrão permite construir diferentes representações de um objeto usando o mesmo código de construção. O padrão Builder simplifica a construção de objetos complexos.	O Builder é representado por uma interface que declara atributos e métodos que são comuns a todos os tipos de construtores. Serão implementados construtores baseados no Builder, que parâmetros que não serão comuns a todos os objetos. Desses construtores resultam os Products. Existe um Director que define a ordem passos de construção. O Cliente utiliza novos objetos criados pelo Director.	Permite construir objetos passo-a-passo. Reutilizar o mesmo código de construção para construir várias representações de produtos. Cumpre o Single Responsibility Principle, permite isolar código de construção complexo da classe onde será usado o produto. Um carro é um objeto complexo que pode ser construído em várias maneiras diferentes. Tornando o Carro numa interface simples, que constitui uma série de métodos comuns a todos os Carros.
O padrão Builder ajuda-nos a resolver o problema de ter algum tipo de classe complexa que possa ter vários atributos opcionais e, consequentemente, os construtores podem chegar a ter demasiados parametros.	A solução é criar uma classe separada que utilize vários métodos para "construir" todos os atributos opcionais da classe complexa.	Dois dos beneficios deste padrão são: poder construir os objetos passo a passo; segue o principio Single Responsibility, pois este isola o código complexo de construção da classe. Um dos exemplos deste padrão é o próprio StringBuilder do java. Este vai construindo a String passo a passo até ficar com uma String bastante mais complexa.
O padrao Builder é um padrão criacional usado no design de software para construir um objeto complexo passo a passo, forma incremental. Ele resolve o problema de criar objetos complexos que podem ter muitas variantes de construção.	O padrao Builder é constituida por: Interface do Builder, define os métodos que serão usados para construir o objeto; Concrete Builders, fornecem a implementação específica dos métodos de construção; Produto, o objeto complexo que é criado passo a passo pelo processo de construção; Director chama os métodos do Builder para construir o objeto; Cliente, associa o objecto builder a um director;	Legibilidade; Flexibilidade; Encadeamento de Métodos; exemplos: APIs e Bibliotecas.
O Builder facilita a construção de objetos complexos separando a criação dos objetos em etapas. É útil quando o objeto é composto por várias partes, até mesmo outros objetos que precisam de ser inicializados. Ao usar este padrão, torna-se o código mais legível e fácil de fazer manutenção.	A estrutura do padrão Builder envolve quatro componentes principais: - A interface Builder: define os métodos para construir as partes de um objeto complexo. - O ConcreteBuilder: implementa a interface Builder e constrói a parte concreta do objeto. - O Director: utiliza a interface Builder para juntar as partes ao produto que se quer criar. - O Product: é o objeto complexo final que resulta deste processo.	Torna o código mais legível e fácil de manter já que aumenta a flexibilidade do código. É mais seguro já que o objeto só é criado depois de todas as etapas estarem concluídas. Toda esta modularidade facilita também a escrita de testes.
It is a pattern that allows the construction of complex objects step by step. It also allows the construction of different types and representations of objects using the same construction code. It solves the problem where you have lots of arguments, sometimes nested, to parse to the constructor. Such initialization code would usually be written inside a enormous constructor with all those attributes, or worse, all over the client code.	It consists of a Builder interface that declares construction steps that are common to all the builders; Concrete Builders which provide different implementations of the construction steps. Those may produce products that don't follow the common interface; Products, that are the resulting objects which may not belong to the same class hierarchy or interface; Director, that defines the order in which to call construction steps, so that it may create and reuse specific configurations of products; Client, by associating one of the builder objects with the director, gets the desired object.	It allows the constructions of objects step-by-step, the reuse of construction code of various different objects, and it isolates complex construction code from the business logic of the product.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

The Builder pattern solves several problems in software design:
- Complex Object Construction: It provides a way to construct complex objects with a large number of optional components or configurations in a clear and flexible manner.
- Step-by-Step Construction: It allows for the construction of an object in a step-by-step process, where different configurations or options can be set at different stages.
- Avoiding Constructors with Multiple Parameters: It helps to avoid the issue of having constructors with too many parameters, which can become unwieldy and error-prone.

The Builder interface declares product construction steps that are common to all types of builders.
Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
Products are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
The Client must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

You can construct objects step-by-step, defer construction steps or run steps recursively.
You can reuse the same construction code when building various representations of products.
You can isolate complex construction code from the business logic of the product.

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

Builders Concretos provém diferentes implementações das etapas de construção. Builders concretos podem produzir produtos que não seguem a interface comum.

Produtos são os objetos resultantes. Produtos construídos por diferentes builders não precisam pertencer a mesma interface ou hierarquia da classe.

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

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.

Construir objetos passo a passo, adiar as etapas de construção ou rodar etapas recursivamente.

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.

Está a ser desenvolvido uma biblioteca para gerar elementos HTML de forma programática. Pode ter diferentes tipos de elementos como , <p>, <a>, etc., cada um com vários atributos, como id, class, href, src, etc. Usando o padrão Builder, é possível criar um construtor para cada tipo de elemento,sendo possível a construção de elementos com facilidade e flexibilidade.

O Builder é um padrão de projeto criacional que permite construir objetos complexos passo a passo. O principal problema resolvido pelo padrão Builder é lidar com a criação de objetos complexos com múltiplos parâmetros ou configurações opcionais. Sem o padrão Builder, os construtores com muitos parâmetros ou construtores telescópicos (com muitos parâmetros opcionais) podem se tornar desajeitados, difíceis de usar e propensos a erros. Além disso, se o objeto sendo criado for imutável, pode ser desafiador definir todos os seus parâmetros de uma vez

It simplifies the process of creating a lot of different complex object. In a normal case you would create a subclass for each different type of complex object or have a class with all the different possible combinations, but not all parameters are needed at the same time.The builder separates the construction of an object step by step with different methods.

The main component is a Builder interface that will be common to all other builders, those said builders with different implementations of the methods in the Builder interface. And a Director class that will be used to call the construction steps so they can be reused and altered.

With the builder pattern you are able to construct different types of object by reusing already existing code, without having to create new subclasses for everything. A possible practical example could be a Builder for a Car class where lets say if you need to create a different type of car.

Evita construtores complexos com demasiados parâmetros, além disso o objeto, depois de ser criado, não pode ser alterado de nenhuma forma. Os dois primeiros pontos apontam para uma quebra de "Readability" e para uma difícil manutenção do código.

Os principais componentes são: Uma interface "Builder" que define os métodos para a criação do objeto; uma ou várias classes "ConcreteBuilder" que implementam a interface e criam o objeto desejado. Cada "ConcreteBuilder" é responsável por juntar as peças necessárias para construir o objeto; a classe "Director" decide qual dos "ConcreteBuilder" a usar na criação do objeto desejado.

Permite a separacao entre a criacao e representacao do objeto
Exemplos: StringBuilder

It solves problems associated with Complex object construction, offers variability and costomization, allows readability and maintainability and extinguish Encapsulation and Separation of Concerns.	Builder - Interface that declares Product construction steps common to all builders. Concrete Builders - Provide different implementations of the construction steps. Products - Resulting objects constructed by different builders. Director - Class that defines the order in which to call construction steps. Client - Associates builder objects with the director.	The Builder pattern allows you to produce different types and representations of an object using the same construction code. The pattern organizes object construction into a set of steps, for example, when creating a house there would be a Builder with various steps like buildWalls, buildDoor, etc.. To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.
Builder é um padrão de creação que consiste em uma classe "Builder" para organizar a construção de um objeto em um conjunto passo-a-passo. É uma solução para classes que possuem vários passos complexos para a instancia-lo, então com o "Builder" estes passos são resumidos e usados apenas quando necessário.	Uma interface "Builder" que declara todos os métodos comuns para qualquer classe builder. Concrete Builders que oferecem diferentes implimentações dos passos de construção para um mesmo objeto, possuí métodos "build" para cada passo. Product a classe que é construída pelos Builders que não precisa pertecer a hierarquia dos builder nem implementar as mesmas interfaces.	Util para criação de objetos que precisam de diferentes fases. Como uma classe Carros que precisa de matrículas, modelos, marcam, pôtencia, etc