**Propósito**

**Builder** es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

**Problema**

Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente, este código de inicialización está sepultado dentro de un monstruoso constructor con una gran cantidad de parámetros. O, peor aún: disperso por todo el código cliente.

Por ejemplo, pensemos en cómo crear un objeto Casa. Para construir una casa sencilla, debemos construir cuatro paredes y un piso, así como instalar una puerta, colocar un par de ventanas y ponerle un tejado. Pero ¿qué pasa si quieres una casa más grande y luminosa, con un jardín y otros extras (como sistema de calefacción, instalación de fontanería y cableado eléctrico)?

La solución más sencilla es extender la clase base Casa y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases. Cualquier parámetro nuevo, como el estilo del porche, exigirá que incrementes esta jerarquía aún más.

Existe otra posibilidad que no implica generar subclases. Puedes crear un enorme constructor dentro de la clase base Casa con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina la necesidad de las subclases, genera otro problema.

En la mayoría de los casos, gran parte de los parámetros no se utilizará, lo que provocará que [**las llamadas al constructor sean bastante feas**](https://refactoring.guru/es/smells/long-parameter-list). Por ejemplo, solo una pequeña parte de las casas tiene piscina, por lo que los parámetros relacionados con piscinas serán inútiles en nueve de cada diez casos.

**Solución**

El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores.

El patrón organiza la construcción de objetos en una serie de pasos (construirParedes, construirPuerta, etc.). Para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, las paredes de una cabaña pueden ser de madera, pero las paredes de un castillo tienen que ser de piedra.

En este caso, podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente. Entonces podemos utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos.

Por ejemplo, imagina un constructor que construye todo de madera y vidrio, otro que construye todo con piedra y hierro y un tercero que utiliza oro y diamantes. Al invocar la misma serie de pasos, obtenemos una casa normal del primer constructor, un pequeño castillo del segundo y un palacio del tercero. Sin embargo, esto sólo funcionaría si el código cliente que invoca los pasos de construcción es capaz de interactuar con los constructores mediante una interfaz común.

**Clase directora**

Puedes ir más lejos y extraer una serie de llamadas a los pasos del constructor que utilizas para construir un producto y ponerlas en una clase independiente llamada directora. La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos.

No es estrictamente necesario tener una clase directora en el programa, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. No obstante, la clase directora puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa.

Además, la clase directora esconde por completo los detalles de la construcción del producto al código cliente. El cliente sólo necesita asociar un objeto constructor con una clase directora, utilizarla para iniciar la construcción, y obtener el resultado del objeto constructor.

**Aplicabilidad**

 Utiliza el patrón Builder para evitar un “constructor telescópico”.

 Digamos que tenemos un constructor con diez parámetros opcionales. Invocar a semejante bestia es poco práctico, por lo que sobrecargamos el constructor y creamos varias versiones más cortas con menos parámetros. Estos constructores siguen recurriendo al principal, pasando algunos valores por defecto a cualquier parámetro omitido.

**class** **Pizza** {

Pizza(**int** size) { ... }

Pizza(**int** size, **boolean** cheese) { ... }

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

// ...

Crear un monstruo semejante sólo es posible en lenguajes que soportan la sobrecarga de métodos, como C# o Java.

El patrón Builder permite construir objetos paso a paso, utilizando tan solo aquellos pasos que realmente necesitamos. Una vez implementado el patrón, ya no hará falta apiñar decenas de parámetros dentro de los constructores.

 Utiliza el patrón Builder cuando quieras que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, casas de piedra y madera).

 El patrón Builder se puede aplicar cuando la construcción de varias representaciones de un producto requiera de pasos similares que sólo varían en los detalles.

La interfaz constructora base define todos los pasos de construcción posibles, mientras que los constructores concretos implementan estos pasos para construir representaciones particulares del producto. Entre tanto, la clase directora guía el orden de la construcción.

 Utiliza el patrón Builder para construir árboles con el patrón [Composite](https://refactoring.guru/es/design-patterns/composite) u otros objetos complejos.

 El patrón Builder te permite construir productos paso a paso. Podrás aplazar la ejecución de ciertos pasos sin descomponer el producto final. Puedes incluso invocar pasos de forma recursiva, lo cual resulta útil cuando necesitas construir un árbol de objetos.

Un constructor no expone el producto incompleto mientras ejecuta los pasos de construcción. Esto evita que el código cliente extraiga un resultado incompleto.

**Pros**

* Puedes construir objetos paso a paso, aplazar pasos de la construcción o ejecutar pasos de forma recursiva.
* Puedes reutilizar el mismo código de construcción al construir varias representaciones de productos.
* *Principio de responsabilidad única*. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.

## Contras

* La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.