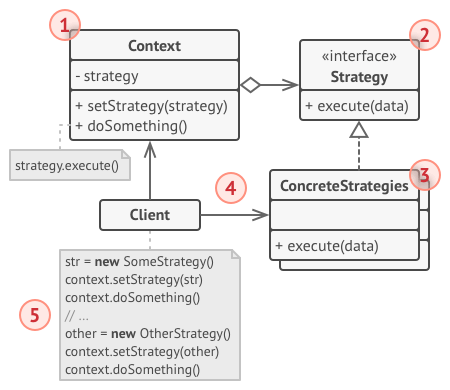
**Strategy Pattern:** es un patrón de diseño de comportamiento que le permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.



1. **Estructura**
   1. El **contexto** mantiene una referencia a una de las estrategias concretas y se comunica con este objeto solo a través de la interfaz de la estrategia.
   2. La interfaz de **estrategia** es común a todas las estrategias concretas. Declara un método que el contexto utiliza para ejecutar una estrategia.
   3. **Las estrategias concretas** implementan diferentes variaciones de un algoritmo que utiliza el contexto.
   4. El contexto llama al método de ejecución en el objeto de estrategia vinculada cada vez que necesita ejecutar el algoritmo. El contexto no sabe con qué tipo de estrategia funciona o cómo se ejecuta el algoritmo.
   5. El **Cliente** crea un objeto de estrategia específico y lo pasa al contexto. El contexto expone un setter que permite a los clientes reemplazar la estrategia asociada con el contexto en tiempo de ejecución.
2. **Pizza estilo paisa**

name = "Pizza estilo paisa";

        dough = "Masa extra gruesa";

        sauce = "Salsa de tomate para pizza";

        toppings.add("tocineta picada");

        toppings.add("chorizo");

        toppings.add("maíz tierno");

        toppings.add("queso mozzarella rayado");

        toppings.add("salsa de tomate");

        toppings.add("pimentón");

        toppings.add("cebolla cortada en medias rodajas");

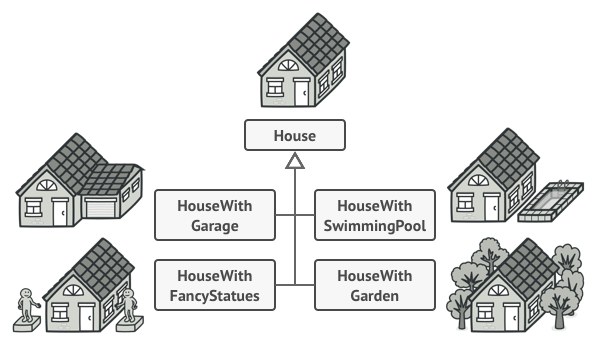
        toppings.add("ajo");

1. **Builder**

es un patrón de diseño creativo que le permite construir objetos complejos paso a paso. El patrón le permite producir diferentes tipos y representaciones de un objeto utilizando el mismo código de construcción.

**Problema**

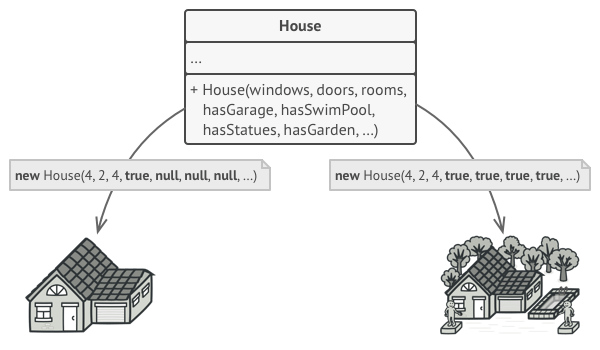
Imagine un objeto complejo que requiere una laboriosa inicialización paso a paso de muchos campos y objetos anidados. Tal código de inicialización generalmente está enterrado dentro de un monstruoso constructor con muchos parámetros. O peor aún: dispersos por todo el código del cliente.



Por ejemplo, pensemos en cómo crear un *House* objeto. Para construir una casa simple, debe construir cuatro paredes y un piso, instalar una puerta, colocar un par de ventanas y construir un techo. Pero, ¿qué pasa si desea una casa más grande y brillante, con un patio trasero y otras cosas buenas (como un sistema de calefacción, plomería y cableado eléctrico)?

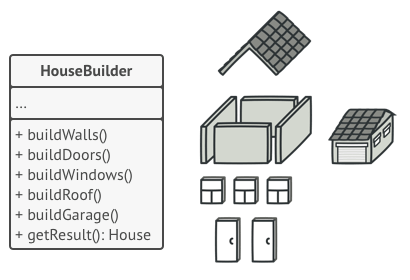
La solución más simple es extender la *House* clase base y crear un conjunto de subclases para cubrir todas las combinaciones de los parámetros. Pero eventualmente terminarás con un número considerable de subclases. Cualquier parámetro nuevo, como el estilo de porche, requerirá aumentar aún más esta jerarquía.

Hay otro enfoque que no implica subclases de cría. Puede crear un constructor gigante directamente en la *House* clase base con todos los parámetros posibles que controlan el objeto de la casa. Si bien este enfoque elimina la necesidad de subclases, crea otro problema.



**Solución:**

El patrón Builder sugiere que extraiga el código de construcción del objeto de su propia clase y lo mueva a objetos separados llamados constructores.



El patrón Builder le permite construir objetos complejos paso a paso. El generador no permite que otros objetos accedan al producto mientras se está construyendo.

El patrón organiza construcción de objetos en un conjunto de pasos ( buildWalls, buildDoor, etc.). Para crear un objeto, ejecuta una serie de estos pasos en un objeto generador. La parte importante es que no necesita llamar a todos los pasos. Puede invocar solo aquellos pasos que son necesarios para producir una configuración particular de un objeto.

Algunos de los pasos de construcción pueden requerir una implementación diferente cuando necesite crear varias representaciones del producto. Por ejemplo, las paredes de una cabaña pueden estar construidas de madera, pero las paredes del castillo deben estar construidas con piedra.

En este caso, puede crear varias clases de constructores diferentes que implementen el mismo conjunto de pasos de construcción, pero de manera diferente. Luego puede usar estos constructores en el proceso de construcción (es decir, un conjunto ordenado de llamadas a los pasos de construcción) para producir diferentes tipos de objetos.

Referencias:

<https://refactoring.guru/design-patterns>

<https://refactoring.guru/design-patterns/builder>