# **Builder**

## **Intención**

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

## **Problema**

Imagine un objeto complejo que requiera una laboriosa inicialización paso a paso de muchos campos y objetos anidados. Dicho código generalmente está enterrado dentro de un constructor con muchos parámetros, o peor aún, dispersos por todo el código del cliente.

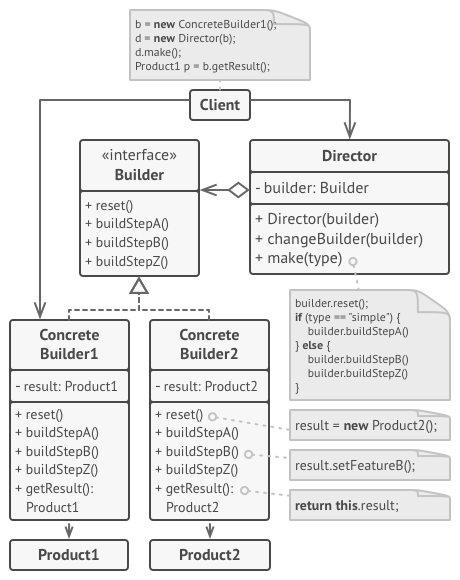
Por ejemplo, pensemos cómo crear un House objeto. Para construir una casa simple, necesitas construir cuatro paredes, instalar una puerta y dos ventanas, construir un techo. Pero, ¿qué pasa si quieres una casa más grande y más luminosa, con un patio y otras cosas?

La solución más simple que viene a la mente es ampliar la House clase base y crear subclases para cubrir todas las combinaciones de los parámetros. El problema obvio con este enfoque es un número considerable de subclases con las que terminará. Cualquier parámetro nuevo, como el estilo de porche, requerirá el crecimiento de esta jerarquía aún más.

Por otro lado, puede crear un constructor gigante en la House clase base que acepte todos los parámetros posibles de la casa resultante. Si bien este enfoque eliminará la necesidad de sub-cadenas, también creará otro problema.

Algunos de estos parámetros no se utilizarán la mayor parte del tiempo, lo que hace que [**las llamadas a los constructores sean bastante feas**](https://refactoring.guru/smells/long-parameter-list) . Por ejemplo, la mayoría de las casas no tienen piscina. Por lo tanto, todos los parámetros relacionados con las piscinas serán inútiles en el 99% de los casos.

## **Solución**



1. **El constructor** declara los pasos necesarios para construir un producto.
2. **Los Constructores Concretos** proporcionan diferentes implementaciones de los pasos de construcción. Los constructores también pueden proporcionar métodos para obtener un resultado de construcción. Este método no se puede declarar dentro de la interfaz del constructor, porque los constructores pueden producir productos que no tienen la interfaz común. Pero si está tratando con productos de una sola jerarquía, este método puede describirse de manera segura
3. **El producto** es un objeto creado como resultado de la construcción. Los constructores pueden producir productos que no pertenecen a la misma jerarquía o interfaz de clase. Es una diferencia clave entre el Constructor y otros patrones de creación.
4. **Director** construye productos usando un objeto Builder. Normalmente, el cliente asigna una instancia de constructor a un director solo una vez a través de los parámetros del constructor. Luego, el director usa ese objeto constructor único para todas las construcciones posteriores. Pero hay una alternativa cuando un constructor pasa al método de producción principal de un director.
5. Tenga en cuenta que el Generador no requiere la creación de una clase de director. La clase de director por separado es útil cuando tiene varias variaciones de producto que requieren diferentes procesos de construcción. Director puede encapsular todo ese código dentro de una sola clase.

## **Pseudocódigo**

Este ejemplo ilustra cómo se utiliza Builder para la construcción paso a paso de automóviles. La clase directora produce varios modelos de automóviles utilizando un conjunto diferente de pasos de construcción. El director trabaja con un objeto de construcción que le fue dado. Esto le permite reutilizar el código de construcción existente para producir manuales de usuario para varios modelos de automóviles. Para lograr esto, solo necesitas una nueva clase de constructor.

Por lo tanto, el patrón de diseño de Builder le permite crear diferentes variaciones de productos, simplemente cambiando la cantidad y el orden de los pasos de construcción. Además, puede producir un tipo de producto completamente diferente utilizando el mismo código de construcción, al enviar una versión alternativa de un objeto de construcción.

// Builder can create different products using the same building

// process.

**class** **Car** **is**

// Can have GPS, trip computer and various numbers of seats.

// Can be a city car, a sports car, or a cabriolet.

**class** **Manual** **is**

// Textual user's manual that corresponds to a particular

// car configuration.

// Builder interface defines all possible ways to configure a

// product.

**interface** **Builder** **is**

**method** reset()

**method** setSeats(...)

**method** setEngine(...)

**method** setTripComputer(...)

**method** setGPS(...)

// Concrete builders implement that interface differently.

**class** **CarBuilder** **implements** Builder **is**

**private** **field** car:Car

**method** reset()

// Put a new Car instance into the "car" field.

**method** setSeats(...) **is**

// Set the number of seats in car.

**method** setEngine(...) **is**

// Install a given engine.

**method** setTripComputer(...) **is**

// Install a trip computer.

**method** setGPS(...) **is**

// Install a global positioning system.

**method** getResult(): Car **is**

// Return the current car object.

// Unlike with other creational patterns, with Builder you can

// construct unrelated products, which do not follow the common

// interface.

**class** **CarManualBuilder** **implements** Builder **is**

**private** **field** manual:Manual

**method** reset()

// Put a new Manual instance into the "manual" field.

**method** setSeats(...) **is**

// Document car seats features.

**method** setEngine(...) **is**

// Add an engine instruction.

**method** setTripComputer(...) **is**

// Add a trip computer instruction.

**method** setGPS(...) **is**

// Add GPS instruction.

**method** getResult(): Manual **is**

// Return the current manual object.

// Director defines the order of building steps. It works with a

// builder object through the common builder interface.

// Therefore it may not know what product is being built.

**class** **Director** **is**

**method** constructSportsCar(builder: Builder) **is**

builder.reset()

builder.setSeats(2)

builder.setEngine(**new** SportEngine())

builder.setTripComputer(**true**)

builder.setGPS(**true**)

// Director gets the concrete builder object from the client

// (application code). That is because application knows better

// which builder it has to use to get a specific product.

**class** **Application** **is**

**method** makeCar **is**

director = **new** Director()

CarBuilder builder = **new** CarBuilder()

director.constructSportsCar(builder)

Car car = builder.getResult()

CarManualBuilder builder = **new** CarManualBuilder()

director.constructSportsCar(builder)

// The final product is often retrieved from a builder

// object, since Director is not aware and not dependent

// on concrete builders and products.

Manual manual = builder.getResult()

## **Aplicabilidad**

**Cuando tienes un constructor "telescópico".**

**No es conveniente llamar a un constructor con una docena de parámetros opcionales. Debe especificar todos los parámetros, incluso si no los necesita.**

**Para aliviar el dolor, uno puede sobrecargar un constructor largo y crear varias versiones más cortas con menos parámetros. Aún llamarán al constructor principal, pero pasarán algunos valores predeterminados a los parámetros omitidos.**

El patrón Builder permite construir objetos paso a paso. Además, puede usar solo los pasos necesarios y omitir los opcionales al crear un objeto simple.

**Cuando su código tiene que crear diferentes representaciones de un producto (por ejemplo, casas de piedra y madera). La construcción del producto tiene pasos similares que difieren en los detalles. Además, aunque los productos pueden ser similares, no es necesario que tengan una clase base o interfaz común.**

 Builder puede usarse para construir diferentes productos usando el mismo proceso de construcción.

Cada producto distinto estará representado por una clase de constructor separada. El código que controla la orden de construcción puede vivir en una sola clase de director.

**Cuando tienes que construir un árbol**[**compuesto**](https://refactoring.guru/design-patterns/composite)**u otro objeto complejo.**

El constructor construye productos paso a paso. Permite la construcción diferida o incluso recursiva que es obligatoria cuando se trabaja con estructuras de árbol. El generador no expone el producto sin terminar mientras ejecuta los pasos de construcción. Esto evita que el código del cliente obtenga resultados corruptos.

## **Cómo implementar**

1. Asegúrese de tener los pasos comunes para construir el producto, así como las variaciones de los pasos que conducen a la creación de varias representaciones de productos.
2. Cree la interfaz de *Builder* y declare los pasos de producción en ella.
3. Cree una clase de *constructor de concreto* para cada una de las representaciones del producto. Implementar sus pasos de construcción.
4. Piensa en crear una clase de *Director*. Sus métodos deberían crear diferentes configuraciones de productos, usando diferentes pasos de la misma instancia del constructor.
5. El código del cliente crea los objetos del *Constructor* y del *Director*. Primero crea una instancia de constructor y luego la pasa al constructor del director o sus métodos de producción.
6. El cliente debe llamar a un método de producción de un objeto *Director* para comenzar el proceso de construcción.
7. El resultado se puede obtener del objeto *Director* solo si todos los productos tienen una interfaz común. En el caso contrario, cada *Constructor* debe tener su propio método de recuperar el resultado.

## **Pros y contras**

**Pros**

* Permite la construcción de productos paso a paso.
* Permite utilizar el mismo código para construir diferentes productos.
* Aísla el código de construcción complejo de la lógica empresarial central de un producto.

**Contras**

Aumenta la complejidad general del código creando múltiples clases adicionales.

## **Ejemplo en Java**

## **builders**

#### **builders/Builder.java:** Common builder interface

**package** refactoring\_guru.builder.example.builders;

**import** refactoring\_guru.builder.example.cars.Type;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Builder interface defines all possible ways to configure a product.*

*\*/*

**public** **interface** **Builder** {

**public** **void** setType(Type type);

**public** **void** setSeats(**int** seats);

**public** **void** setEngine (Engine engine);

**public** **void** setTransmission(Transmission transmission);

**public** **void** setTripComputer(TripComputer tripComputer);

**public** **void** setGPSNavigator(GPSNavigator gpsNavigator);

}

#### **builders/CarBuilder.java:** Builder of car

**package** refactoring\_guru.builder.example.builders;

**import** refactoring\_guru.builder.example.cars.Car;

**import** refactoring\_guru.builder.example.cars.Type;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Concrete builders implement steps defined in the common interface.*

*\*/*

**public** **class** **CarBuilder** **implements** Builder {

**private** Type type;

**private** **int** seats;

**private** Engine engine;

**private** Transmission transmission;

**private** TripComputer tripComputer;

**private** GPSNavigator gpsNavigator;

**@Override**

**public** **void** setType(Type type) {

**this**.type = type;

}

**@Override**

**public** **void** setSeats(**int** seats) {

**this**.seats = seats;

}

**@Override**

**public** **void** setEngine(Engine engine) {

**this**.engine = engine;

}

**@Override**

**public** **void** setTransmission(Transmission transmission) {

**this**.transmission = transmission;

}

**@Override**

**public** **void** setTripComputer(TripComputer tripComputer) {

**this**.tripComputer = tripComputer;

}

**@Override**

**public** **void** setGPSNavigator(GPSNavigator gpsNavigator) {

**this**.gpsNavigator = gpsNavigator;

}

**public** Car getResult() {

**return** **new** Car(type, seats, engine, transmission, tripComputer, gpsNavigator);

}

}

#### **builders/CarManualBuilder.java:** Builder of a car manual

**package** refactoring\_guru.builder.example.builders;

**import** refactoring\_guru.builder.example.cars.Manual;

**import** refactoring\_guru.builder.example.cars.Type;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Unlike other creational patterns, Builder can construct unrelated products,*

*\* which don't have the common interface.*

*\**

*\* In this case we build a user manual for a car, using the same steps as we*

*\* built a car. This allows to produce manuals for specific car models,*

*\* configured with different features.*

*\*/*

**public** **class** **CarManualBuilder** **implements** Builder{

**private** Type type;

**private** **int** seats;

**private** Engine engine;

**private** Transmission transmission;

**private** TripComputer tripComputer;

**private** GPSNavigator gpsNavigator;

**@Override**

**public** **void** setType(Type type) {

**this**.type = type;

}

**@Override**

**public** **void** setSeats(**int** seats) {

**this**.seats = seats;

}

**@Override**

**public** **void** setEngine(Engine engine) {

**this**.engine = engine;

}

**@Override**

**public** **void** setTransmission(Transmission transmission) {

**this**.transmission = transmission;

}

**@Override**

**public** **void** setTripComputer(TripComputer tripComputer) {

**this**.tripComputer = tripComputer;

}

**@Override**

**public** **void** setGPSNavigator(GPSNavigator gpsNavigator) {

**this**.gpsNavigator = gpsNavigator;

}

**public** Manual getResult() {

**return** **new** Manual(type, seats, engine, transmission, tripComputer, gpsNavigator);

}

}

## **Cars**

#### **cars/Car.java:** Car product

**package** refactoring\_guru.builder.example.cars;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Car is a product class.*

*\*/*

**public** **class** **Car** {

**private** **final** Type type;

**private** **final** **int** seats;

**private** **final** Engine engine;

**private** **final** Transmission transmission;

**private** **final** TripComputer tripComputer;

**private** **final** GPSNavigator gpsNavigator;

**private** **double** fuel = 0;

**public** Car(Type type, **int** seats, Engine engine, Transmission transmission,

TripComputer tripComputer, GPSNavigator gpsNavigator) {

**this**.type = type;

**this**.seats = seats;

**this**.engine = engine;

**this**.transmission = transmission;

**this**.tripComputer = tripComputer;

**this**.tripComputer.setCar(**this**);

**this**.gpsNavigator = gpsNavigator;

}

**public** Type getType() {

**return** type;

}

**public** **double** getFuel() {

**return** fuel;

}

**public** **void** setFuel(**double** fuel) {

**this**.fuel = fuel;

}

**public** **int** getSeats() {

**return** seats;

}

**public** Engine getEngine() {

**return** engine;

}

**public** Transmission getTransmission() {

**return** transmission;

}

**public** TripComputer getTripComputer() {

**return** tripComputer;

}

**public** GPSNavigator getGpsNavigator() {

**return** gpsNavigator;

}

}

#### **cars/Manual.java:** Manual product

**package** refactoring\_guru.builder.example.cars;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Car manual is another product. Note that it does not have the same ancestor*

*\* as a Car. They are not related.*

*\*/*

**public** **class** **Manual** {

**private** **final** Type type;

**private** **final** **int** seats;

**private** **final** Engine engine;

**private** **final** Transmission transmission;

**private** **final** TripComputer tripComputer;

**private** **final** GPSNavigator gpsNavigator;

**public** Manual(Type type, **int** seats, Engine engine, Transmission transmission,

TripComputer tripComputer, GPSNavigator gpsNavigator) {

**this**.type = type;

**this**.seats = seats;

**this**.engine = engine;

**this**.transmission = transmission;

**this**.tripComputer = tripComputer;

**this**.gpsNavigator = gpsNavigator;

}

**public** **String** print() {

**String** info = "";

info += "Type of car: " + type + "\n";

info += "Count of seats: " + seats + "\n";

info += "Engine: volume - " + engine.getVolume() + "; mileage - " + engine.getMileage() + "\n";

info += "Transmission: " + transmission + "\n";

**if** (**this**.tripComputer != **null**) {

info += "Trip Computer: Functional" + "\n";

} **else** {

info += "Trip Computer: N/A" + "\n";

}

**if** (**this**.gpsNavigator != **null**) {

info += "GPS Navigator: Functional" + "\n";

} **else** {

info += "GPS Navigator: N/A" + "\n";

}

**return** info;

}

}

#### **cars/Type.java**

**package** refactoring\_guru.builder.example.cars;

**public** **enum** **Type** {

CITY\_CAR, SPORTS\_CAR, SUV;

}

## **components**

#### **components/Engine.java:** Product feature 1

**package** refactoring\_guru.builder.example.components;

*/\*\**

*\* Just another feature of a car.*

*\*/*

**public** **class** **Engine** {

**private** **final** **double** volume;

**private** **double** mileage;

**private** **boolean** started;

**public** Engine(**double** volume, **double** mileage) {

**this**.volume = volume;

**this**.mileage = mileage;

}

**public** **void** on() {

started = **true**;

}

**public** **void** of() {

started = **false**;

}

**public** **boolean** isStarted() {

**return** started;

}

**public** **void** go(**double** mileage) {

**if** (started) {

**this**.mileage += mileage;

} **else** {

System.err.println("Cannot go(), you must start engine first!");

}

}

**public** **double** getVolume() {

**return** volume;

}

**public** **double** getMileage() {

**return** mileage;

}

}

#### **components/GPSNavigator.java:** Product feature 2

**package** refactoring\_guru.builder.example.components;

*/\*\**

*\* Just another feature of a car.*

*\*/*

**public** **class** **GPSNavigator** {

**private** **String** route;

**public** GPSNavigator() {

**this**.route = "221b, Baker Street, London to Scotland Yard, 8-10 Broadway, London";

}

**public** GPSNavigator(**String** manualRoute) {

**this**.route = manualRoute;

}

**public** **String** getRoute() {

**return** route;

}

}

#### **components/Transmission.java:** Product feature 3

**package** refactoring\_guru.builder.example.components;

*/\*\**

*\* Just another feature of a car.*

*\*/*

**public** **enum** **Transmission** {

SINGLE\_SPEED, MANUAL, AUTOMATIC, SEMI\_AUTOMATIC;

}

#### **components/TripComputer.java:** Product feature 4

**package** refactoring\_guru.builder.example.components;

**import** refactoring\_guru.builder.example.cars.Car;

*/\*\**

*\* Just another feature of a car.*

*\*/*

**public** **class** **TripComputer** {

**private** Car car;

**public** **void** setCar(Car car) {

**this**.car = car;

}

**public** **void** showFuelLevel() {

System.out.println("Fuel level: " + car.getFuel());

}

**public** **void** showStatus() {

**if** (**this**.car.getEngine().isStarted()) {

System.out.println("Car is started");

} **else** {

System.out.println("Car isn't started");

}

}

}

## **director**

#### **director/Director.java:** Director controls builders

**package** refactoring\_guru.builder.example.director;

**import** refactoring\_guru.builder.example.cars.Type;

**import** refactoring\_guru.builder.example.components.Engine;

**import** refactoring\_guru.builder.example.components.GPSNavigator;

**import** refactoring\_guru.builder.example.components.Transmission;

**import** refactoring\_guru.builder.example.builders.Builder;

**import** refactoring\_guru.builder.example.components.TripComputer;

**import** refactoring\_guru.builder.example.components.\*;

*/\*\**

*\* Director defines the order of building steps. It works with a builder object*

*\* through common Builder interface. Therefore it may not know what product is*

*\* being built.*

*\*/*

**public** **class** **Director** {

**public** **void** constructSportsCar(Builder builder) {

builder.setType(Type.SPORTS\_CAR);

builder.setSeats(2);

builder.setEngine(**new** Engine(3.0, 0));

builder.setTransmission(Transmission.SEMI\_AUTOMATIC);

builder.setTripComputer(**new** TripComputer());

builder.setGPSNavigator(**new** GPSNavigator());

}

**public** **void** constructCityCar(Builder builder) {

builder.setType(Type.CITY\_CAR);

builder.setSeats(2);

builder.setEngine(**new** Engine(1.2, 0));

builder.setTransmission(Transmission.AUTOMATIC);

builder.setTripComputer(**new** TripComputer());

builder.setGPSNavigator(**new** GPSNavigator());

}

**public** **void** constructSUV(Builder builder) {

builder.setType(Type.SUV);

builder.setSeats(4);

builder.setEngine(**new** Engine(2.5, 0));

builder.setTransmission(Transmission.MANUAL);

builder.setGPSNavigator(**new** GPSNavigator());

}

}

#### **Demo.java:** Client code

**package** refactoring\_guru.builder.example;

**import** refactoring\_guru.builder.example.builders.CarBuilder;

**import** refactoring\_guru.builder.example.cars.Car;

**import** refactoring\_guru.builder.example.cars.Manual;

**import** refactoring\_guru.builder.example.director.Director;

**import** refactoring\_guru.builder.example.builders.CarManualBuilder;

*/\*\**

*\* Demo class. Everything comes together here.*

*\*/*

**public** **class** **Demo** {

**public** **static** **void** main(**String**[] args) {

Director director = **new** Director();

*// Director gets the concrete builder object from the client*

*// (application code). That's because application knows better which*

*// builder to use to get a specific product.*

CarBuilder builder = **new** CarBuilder();

director.constructSportsCar(builder);

*// The final product is often retrieved from a builder object, since*

*// Director is not aware and not dependent on concrete builders and*

*// products.*

Car car = builder.getResult();

System.out.println("Car built:\n" + car.getType());

CarManualBuilder manualBuilder = **new** CarManualBuilder();

*// Director may know several building recipes.*

director.constructSportsCar(manualBuilder);

Manual carManual = manualBuilder.getResult();

System.out.println("\nCar manual built:\n" + carManual.print());

}

}

#### **OutputDemo.txt:** Execution results

Car built:

SPORTS\_CAR

Car manual built:

Type of car: SPORTS\_CAR

Count of seats: 2

Engine: volume - 3.0; mileage - 0.0

Transmission: SEMI\_AUTOMATIC

Trip Computer: Functional

GPS Navigator: Functional

# **Referencia**

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