

Blog Bitix

[Java](#)[GNU/Linux](#)[JavaScript](#)[Tapestry](#)[Archivo y hemeroteca](#)[Enlaces](#)[Acerca de...](#)

Ejemplo del patrón de diseño Builder

Escrito por [picodotdev](#) el 27/09/2015, actualizado el 10/11/2015.

[java](#) [planeta-codigo](#) [programacion](#)

[Enlace permanente](#) [Comentarios](#)

Construir objetos es una tarea básica en los lenguajes orientados a objetos. En Java, las instancias de una clase se crean con la palabra clave reservada *new* y un método especial llamado constructor. Al diseñar una clase debemos tener algunas cuestiones para evitar varios constructores *telescopicos*, evitar constructores que son combinación de varios argumentos opcionales y permitir obtener instancias de objetos con estado válido. Si se nos presentan estas situaciones podemos usar el patrón de diseño *Builder* que consiste en básicamente en una clase especializada en construir instancias de otra clase que podemos hacer

usable con una API fluida y alguna cosa más deseable que explico en el artículo.

Al escribir los métodos constructores de instancias de una clase puede ocurrirnos que algunos de ellos tienen una lista larga de argumentos (cuatro o más parámetros puede considerarse larga) o el caso de que otros algunos argumentos son opcionales. En el caso de una lista larga de argumentos algunos puedan tomar valores por defecto creando métodos telescópicos (donde hay varios constructores y cada uno solo añade un nuevo argumento al anterior), en el caso de argumentos opcionales nos obliga a crear un constructor por cada combinación de argumentos, peor aún, ambas cosas se pueden producir a la vez.



Por ejemplo, supongamos que tenemos una entidad de dominio *Usuario* en la que el correo electrónico es requerido siendo opcionales su nombre, apellidos teléfono o dirección. Sin usar el patrón de diseño *Builder* probablemente tendríamos los siguientes constructores o tener solo el último de ellos y en los no necesarios usar como valor del argumento *null*.

```
1 // Constructores con el problema de ser telescópicos y ser múltiples por combinación de pa
2 public Usuario(String email)
3 public Usuario(String email, String nombre, String apellidos)
4 public Usuario(String email, String telefono)
5 public Usuario(String email, String direccion)
6 public Usuario(String email, String nombre, String apellidos, String telefono)
7 public Usuario(String email, String nombre, String apellidos, String direccion)
```

```
8 public Usuario(String email, String telefono, String direccion)
9 public Usuario(String email, String nombre, String apellidos, String telefono, String di
```

Usuario-1.java

Como vemos no son pocos constructores debido a las combinaciones de los parámetros opcionales, esta forma requiere una buena cantidad de líneas de código y si decidiésemos escribir solo el constructor con todos los parámetros al usarlo tendremos dificultades para saber a que argumento responde cada variable y probablemente deberemos consultar la firma del constructor para saber que lugar ocupa cada argumento, esto dificulta la legibilidad.

```
1 // Uso de un constructor, ¿a que argumento corresponde cada dato?
2 new Usuario("nombre.apellido@gmail.com", "Nombre", "Apellido", "555123456", "c\\ Rue el
```

Usuario-2.java

En este caso solo hay tres argumentos opcionales si hubiera más el número de combinaciones y por tanto de constructores aumentaría considerablemente. Puede que en vez de usar constructores usemos un método *set* de JavaBean de forma que tengamos un solo constructor y múltiples métodos *set* o un constructor con los argumentos requeridos y un *set* por cada argumento opcional.

```
1 // Siguiendo las convenciones de Los JavaBeans la clase Usuario deja de ser inmutable
2 public Usuario()
3 public Usuario(String email)
4 public setEmail(String email)
5 public setNombre(String nombre, String apellidos)
6 public setTelefono(String telefono)
7 public setDireccion(String direccion)
```

Usuario-3.java

Sin embargo, esta solución aunque permite reducir el número de constructores también tiene problemas, uno de ellos es que el constructor y los *set* no obligan a crear un objeto con estado consistente o válido, otro es que usando los *set* de los JavaBean nos impide hacer el objeto inmutable, si no es devolviendo una nueva instancia, que con las **nuevas características funcionales añadidas en Java 8** y en la programación concurrente es deseable.

El patrón de diseño *Builder*

El patrón de diseño *Builder* es un patrón de diseño clasificado en los creacionales que se encarga de la creación de instancias de clases. Sus ventajas son que solucionan el problema de los constructores telescópicos y combinación de argumentos es usar el patrón de diseño *Builder*, además permite crear objetos complejos de forma flexible en varios pasos con propiedades opcionales.

Al igual que el patrón de diseño *Factory* se encarga de crear instancias de una clase, sin embargo, tiene algunas diferencias como que el patrón *Factory* crea las instancias en un único paso cuando el *Builder* puede crear las instancias en varios pasos, el patrón *Builder* es más complejo pero más flexible. Otra diferencia es que el patrón de diseño *Builder* tiene estado y en el *Factory* no siempre es necesario, esto hace que la instancia de clase *Builder* no se pueda compartir ni utilizar para crear otras instancias.

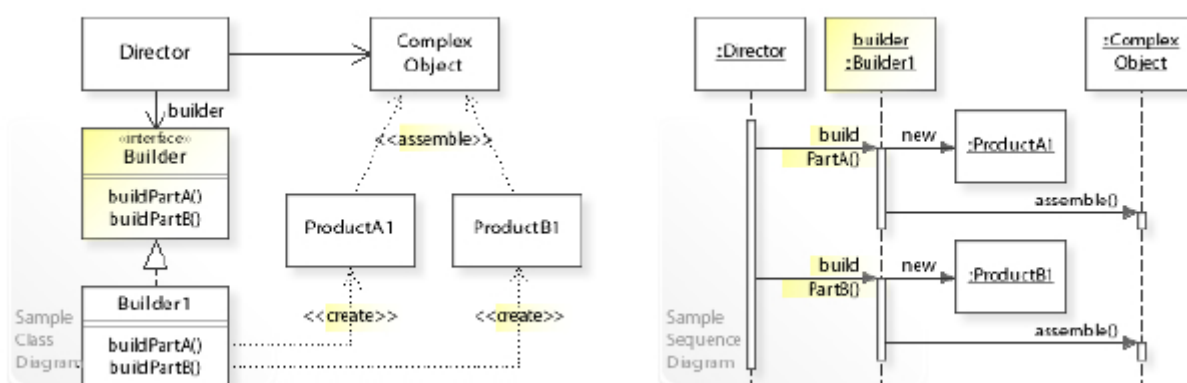


Diagrama de clases del patrón de diseño Builder

Ejemplo del patrón de diseño *Builder*

Empleando el mismo caso que los anteriores de la siguiente forma.

```
1 package io.github.picodotdev.pattern.builder;
2
3 public class Usuario {
4
5     private String email;
6     private String nombre;
7     private String apellidos;
8     private String telefono;
9     private String direccion;
10
11     private Usuario() {
12     }
13 }
```

```

14     Usuario(UsuarioBuilder builder) {
15         if (builder.getEmail() == null) {
16             throw new IllegalArgumentException("email es requerido");
17         }
18         this.email = builder.getEmail();
19         this.nombre = builder.getNombre();
20         this.apellidos = builder.getApellidos();
21         this.telefono = builder.getTelefono();
22         this.direccion = builder.getDireccion();
23     }
24 }

```

Usuario.java

```

1 package io.github.picodotdev.pattern.builder;
2
3 public class UsuarioBuilder {
4
5     private String email;
6     private String nombre;
7     private String apellidos;
8     private String telefono;
9     private String direccion;
10
11     public UsuarioBuilder() {
12     }
13
14     public UsuarioBuilder email(String email) {
15         this.email = email;
16         return this;
17     }
18
19     public UsuarioBuilder nombre(String nombre, String apellidos) {
20         this.nombre = nombre;
21         this.apellidos = apellidos;
22         return this;
23     }
24
25     public UsuarioBuilder telefono(String telefono) {
26         this.telefono = telefono;
27         return this;
28     }
29
30     public UsuarioBuilder direccion(String direccion) {
31         this.direccion = direccion;
32         return this;
33     }
34
35     public Usuario build() {
36         return new Usuario(this);
37     }
38

```

```

39 // Getters
40 public String getEmail() {
41     return email;
42 };
43
44 public String getNombre() {
45     return nombre;
46 };
47
48 public String getApellidos() {
49     return apellidos;
50 };
51
52 public String getTelefono() {
53     return telefono;
54 };
55
56 public String getDireccion() {
57     return direccion;
58 };
59 }

```

UsuarioBuilder.java

Su uso sería de la siguiente manera algo más autoexplicativa y legible que la opción de usar constructores.

```

1 package io.github.picodotdev.pattern.builder;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Usuario usuario = new UsuarioBuilder()
7             .email("nombre.apellido@gmail.com")
8             .nombre("Nombre", "Apellido")
9             .telefono("555123456")
10            .direccion("c\\ Rue el Percebe 13").build();
11     }
12 }

```

Main.java

La instancia de la clase *UsuarioBuilder* en su uso recoge los datos usando una API fluida, el método *build* es el que construye la instancia del usuario mediante el constructor con visibilidad de paquete en el que se valida que los datos al construir el objeto *Usuario* sean válidos, en este caso que el *email* es requerido.

En el libro [Effective Java](#) en el *Item #2* se comenta más detalladamente este patrón junto a otra buena colección de cosas sobre los constructores y más cosas sobre Java, es uno en mi lista de [8+ libros recomendables para mejorar como programadores](#).

En el apartado de referencia puedes encontrar más artículos que he escrito sobre otros patrones de diseño.

Referencia:

- [8+ libros para mejorar como programadores](#)
- [Patrones de diseño en la programación orientada a objetos](#)
- [Ejemplo del patrón de diseño *Command*](#)
- [Ejemplo del patrón de diseño *State*](#)
- [Patrón múltiples vistas de un mismo dato en Tapestry](#)
- [Novedades y nuevas características de Java 8](#)

Este artículo incluye algunos enlaces de afiliado como Amazon que considero relevante en el contenido del artículo y para el lector. En caso de hacer una compra a través de estos enlaces recibo una pequeña comisión, sin que afecte al precio del producto, que me ayuda a seguir publicando nuevos artículos y realizar [pequeñas donaciones a proyectos de software libre](#). En caso de que el artículo te haya resultado de interés y útil considera realiza la compra a través de alguno de los enlaces de afiliado del artículo.

.....

Este artículo forma parte de la serie **java-patron-diseno**:

1. [Ejemplo del patrón de diseño No Operation](#)
2. [Ejemplo del patrón de diseño Builder](#)
3. [El patrón de diseño Observer y una forma de implementarlo en Java](#)
4. [Ejemplo de máquina de estados con Spring Statemachine](#)
5. [El patrón de diseño Specification, ejemplo de implementación y uso en JPA con Spring Data](#)
6. [5 formas de implementar el patrón Singleton en Java](#)
7. [El patrón de diseño Factory, ventajas sobre new y diferencias con Builder](#)

Comparte el artículo:



Blog Bitix

Blog sobre el lenguaje de programación Java y la distribución GNU/Linux que uso habitualmente, Arch Linux, lo que aprendo sobre el software libre, la programación web y otros temas relacionados con la tecnología y la informática. El contenido puede contener trazas de asuntos fuera de tema.

Publicando de uno a tres artículos únicos a la semana desde el año 2010.

[Java](#)

[GNU/Linux](#)

[JavaScript](#)

[Tapestry](#)

[Archivo y hemeroteca](#)

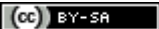
[Enlaces](#)

[Publicidad](#)

[Donaciones](#)

[Acerca de...](#)



Copyleft © 2021 - 



Blog Bitix by [pico.dev](#) is licensed under a [Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional License](#).

Powered by [Hugo](#) and [GitHub Pages](#). Background patterns from [Subtle Patterns](#).