



Patrones de diseño

Randall Corella

Patrones de diseño

- Un patrón de diseño es la solución de un problema de diseño cuya efectividad ha sido verificada por resolver problemas similares anteriormente.
- Estos patrones deben ser reutilizables en diferentes diseños y en distintas circunstancias.





Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de software.
- Evitar la reiteración de búsqueda o creación de diseños con soluciones previas.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar la creación de los diseños.



3 tipos

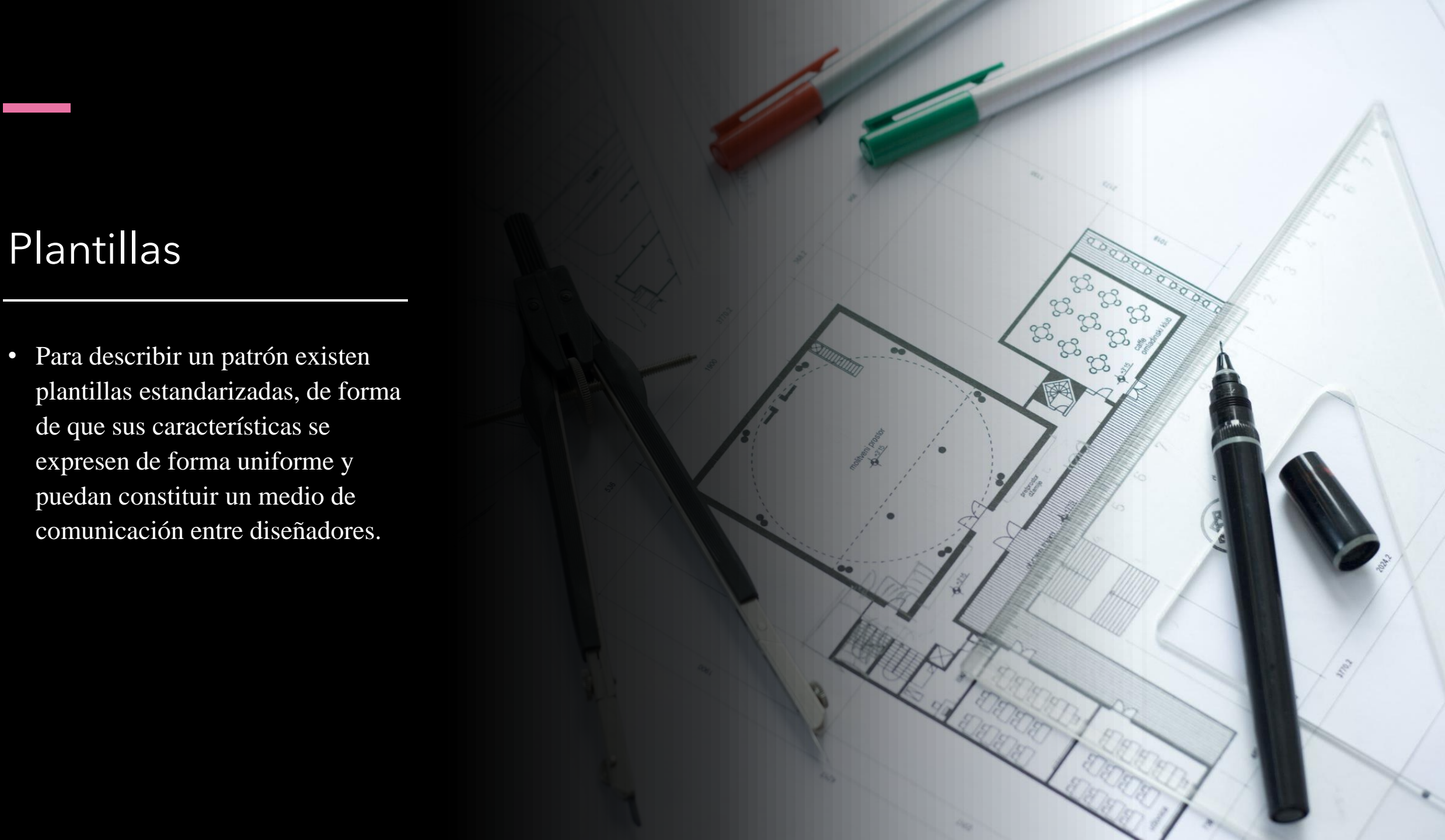
Patrones
Creacionales

Patrones
Estructurales

Patrones de
Comportamiento.

Plantillas

- Para describir un patrón existen plantillas estandarizadas, de forma de que sus características se expresen de forma uniforme y puedan constituir un medio de comunicación entre diseñadores.



A background image showing a grid of colorful tiles, each with a multiplication problem (e.g., 1x1, 2x2, 3x3, etc.) written on it. The tiles are arranged in a grid that recedes into the distance, creating a sense of depth. The colors of the tiles vary, including shades of purple, blue, green, yellow, and orange.

Plantilla

- **Nombre del patrón:** Nombre estándar del patrón.
- **Clasificación del patrón:** Creacional, estructural o de comportamiento.
- **Motivación:** Escenario de ejemplo.
- **Aplicabilidad:** Usos comunes y criterios de aplicabilidad.
- **Estructura:** Diagramas de clases oportunos para describir las clases que intervienen en el patrón.
- **Participantes:** Enumeración y descripción de las entidades abstractas que conforman el patrón.
- **Colaboraciones:** Explicación de las interrelaciones entre los participantes.
- **Consecuencias:** Consecuencias positivas o negativas derivadas de la aplicación del patrón.
- **Implementación:** Técnicas y recomendaciones de cara a la implementación del patrón.
- **Código de ejemplo:** código fuente de ejemplo.
- **Usos conocidos:** Ejemplos de sistemas reales que utilizan el patrón.
- **Patrones relacionados:** referencias cruzadas con otros patrones.

Patrón de diseño de prototipo

- Es un patrón de diseño creacional, que se enfoca en la creación de objetos a partir de un modelo. Se especifica un modelo prototípico genérico del cual a partir de este se crearán nuevos objetos copiando este.
- Es un diseño bastante simple, ya que su concepto es hacer una copia exacta de otro objeto, permite crear objetos prediseñados sin la necesidad de detalles de creación.





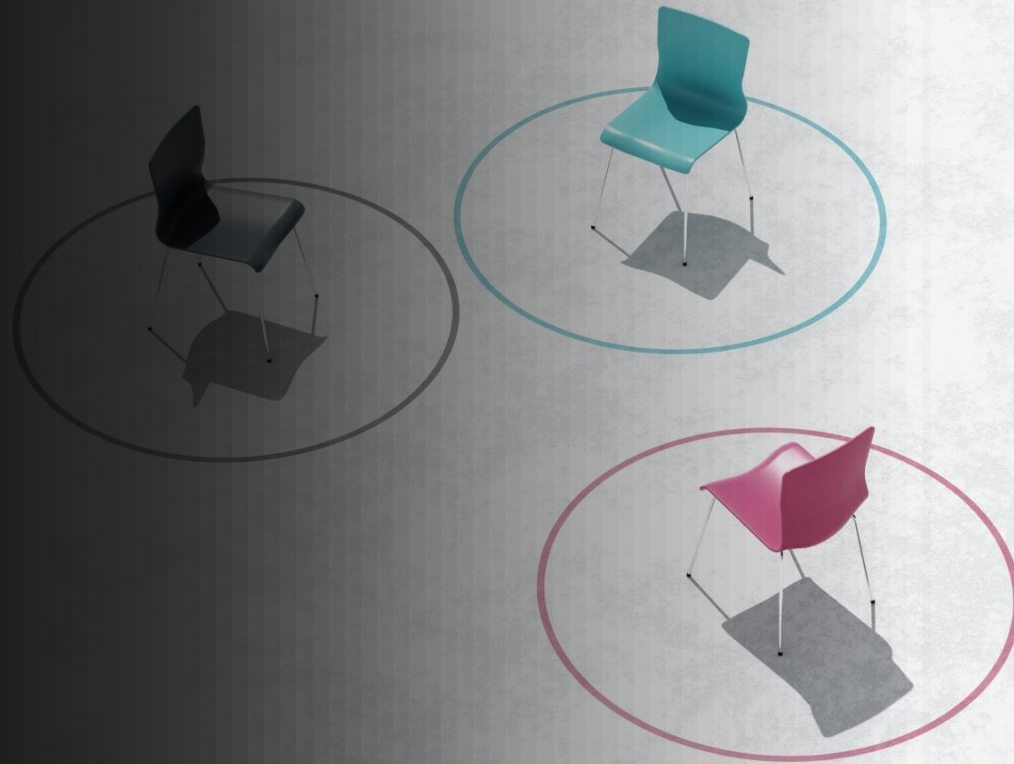
¿Cuándo se utiliza?

- La programación actual se ve afectada directamente por los costos. Por lo que el ahorro es un gran problema cuando hablamos de los recursos de una computadora. Los programadores están haciendo su mejor esfuerzo para mejorar el rendimiento de los programas. Cuando se habla de objetos en los que se puede encontrar una forma óptima de instanciarlos, es donde entra en juego la clonación. Si el coste de creación de un objeto consume muchos recursos es donde decidimos clonarlo. En este caso es donde entra el patrón de diseño, este nos permite clonar el objeto sin la necesidad de conocer los detalles de la clase o el objeto.



Clonación profunda:

- Esta clonación se realiza cuando el objeto que se quiere clonar tiene como atributos otros objetos (1 o más) que se deben clonar de igual manera, para retornar una clonación completa del objeto y no referencias a otros ya existentes.



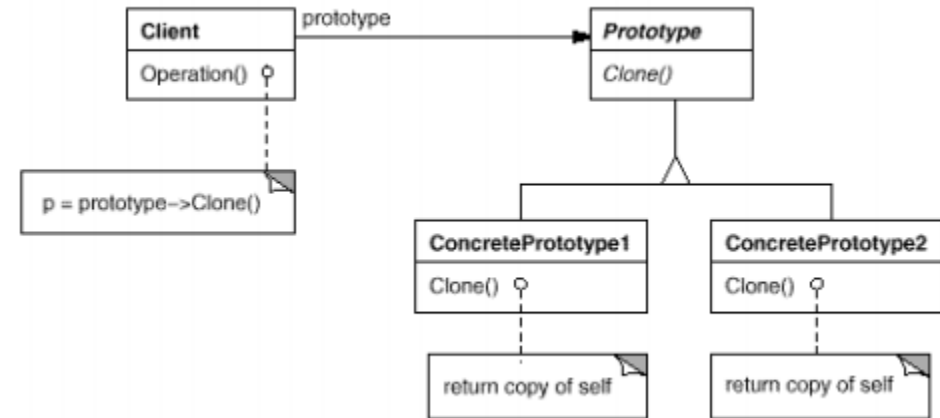
Clonación superficial:

- Por otro lado, esta clonación es aquella que se realiza cuando el objeto que se quiere clonar no posee otros que se deban copiar (tales como Integer, Char, Bool). Se realiza a nivel de bits.

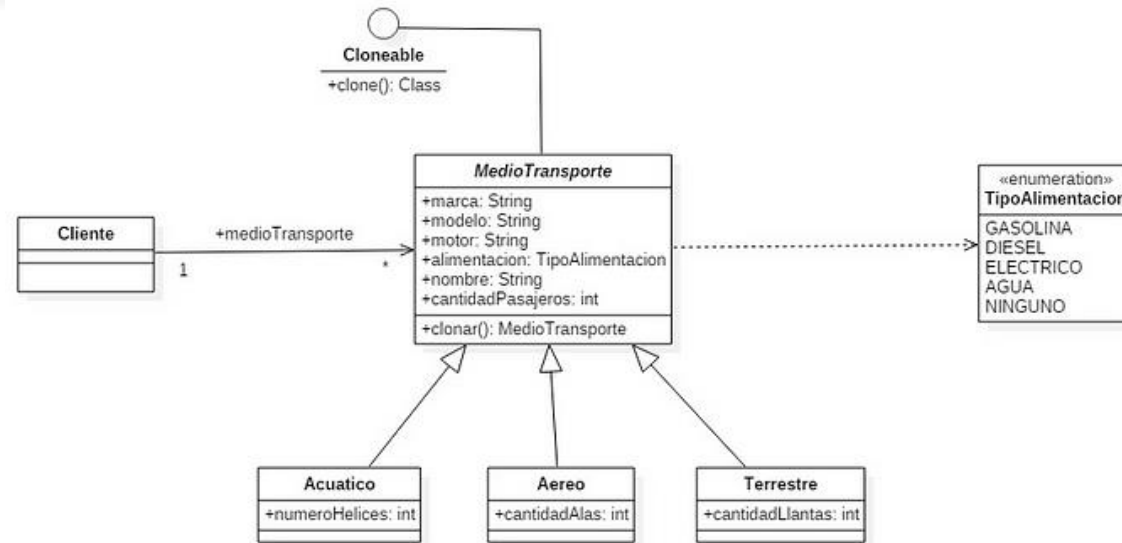


¿Cómo se implementa?

- La idea de este patrón es que los objetos que necesiten ser clonados implementen una interfaz que tenga este método. O bien heredar de una clase abstracta que tenga el método abstracto clonar () para que lo desarrollen sus hijos.



Ejemplo



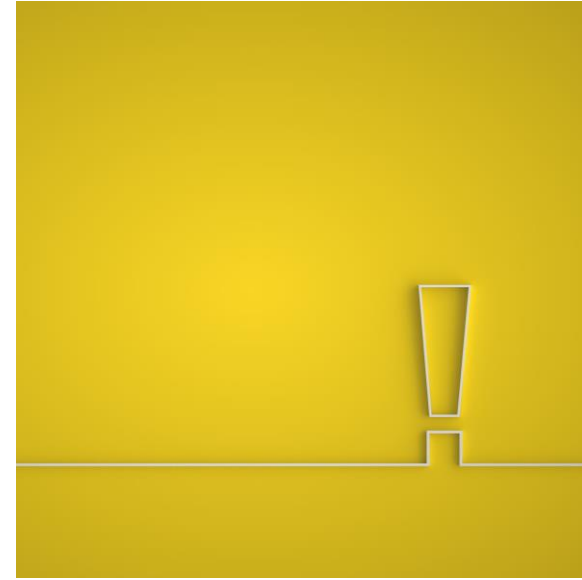
Patrón singleton

- El patrón singleton, o singleton pattern en inglés, pertenece a la categoría de **patrones creativos** dentro del grupo de los patrones de diseño. También se le conoce simplemente como “singleton”. El propósito de este patrón es evitar que sea creado más de un objeto por clase. Esto se logra creando el objeto deseado en una clase y recuperándolo como una instancia estática. El singleton es uno de los **patrones más simples**, pero **más poderosos** en el desarrollo de software.



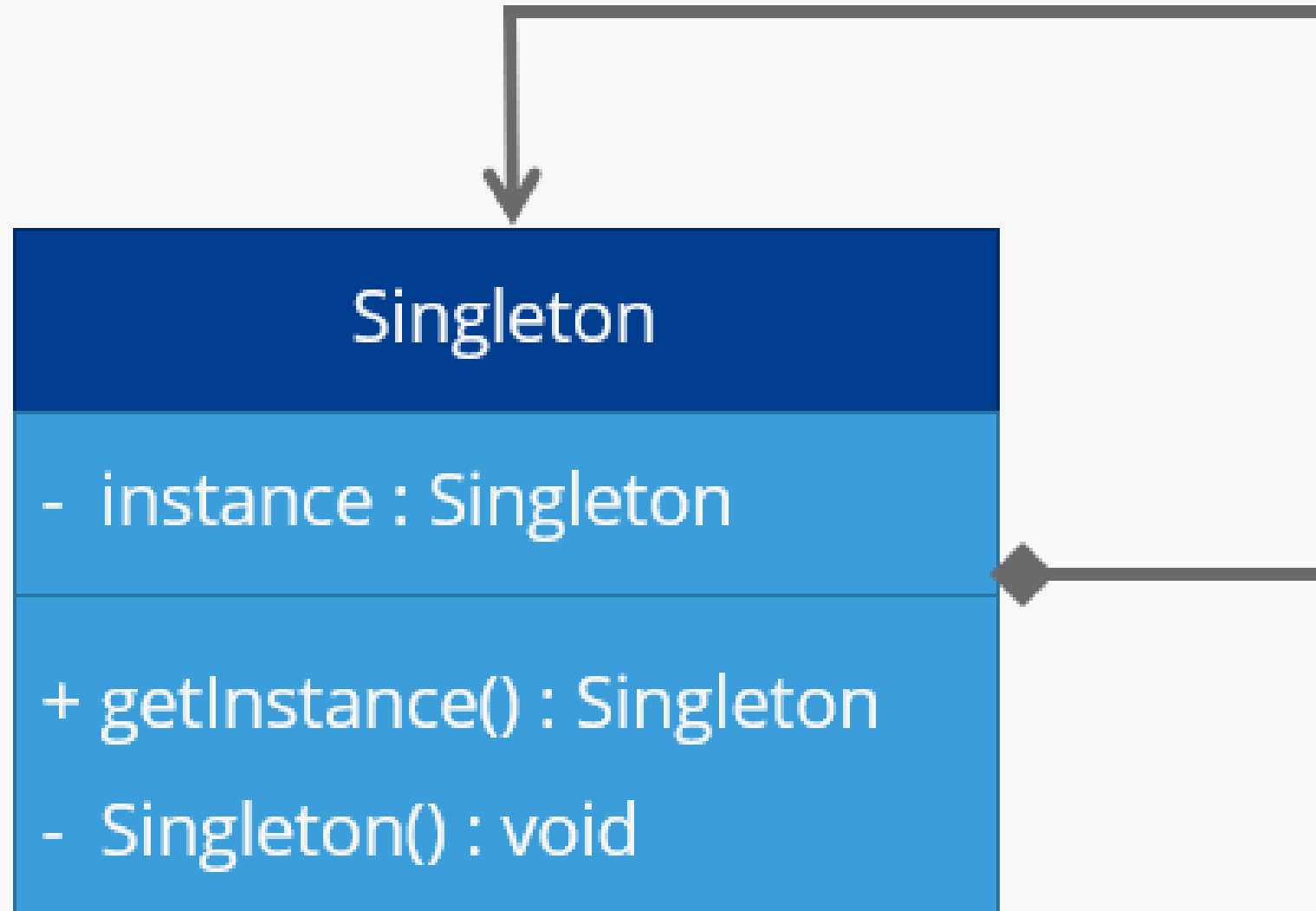
Características principales:

- Si se utiliza el patrón singleton para crear **una instancia** de una clase, entonces el patrón se asegura de que realmente sólo permanezca con esta instancia única. El singleton hace que esta clase de software sea accesible globalmente. En los diferentes lenguajes de programación, hay diferentes métodos para lograrlo. Para asegurarse de que permanezca con una sola instancia única, se debe impedir que los usuarios creen nuevas instancias. Esto se logra mediante el constructor, declarando el patrón como “**privado**”. Esto significa que sólo el código en el singleton puede instanciar el singleton en sí mismo. Por lo tanto, esto garantiza que sólo un mismo objeto puede llegar al usuario. Si esta instancia ya existe, no se crea ninguna nueva instancia.



```
public class Singleton {  
    private static Singleton instance; // protected from external access and static  
    private Singleton() {} // private constructor with external access protection  
    public static getInstance() { // public method, call out by code  
        if (instance == null) { // only if no instance exists, then create a new  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

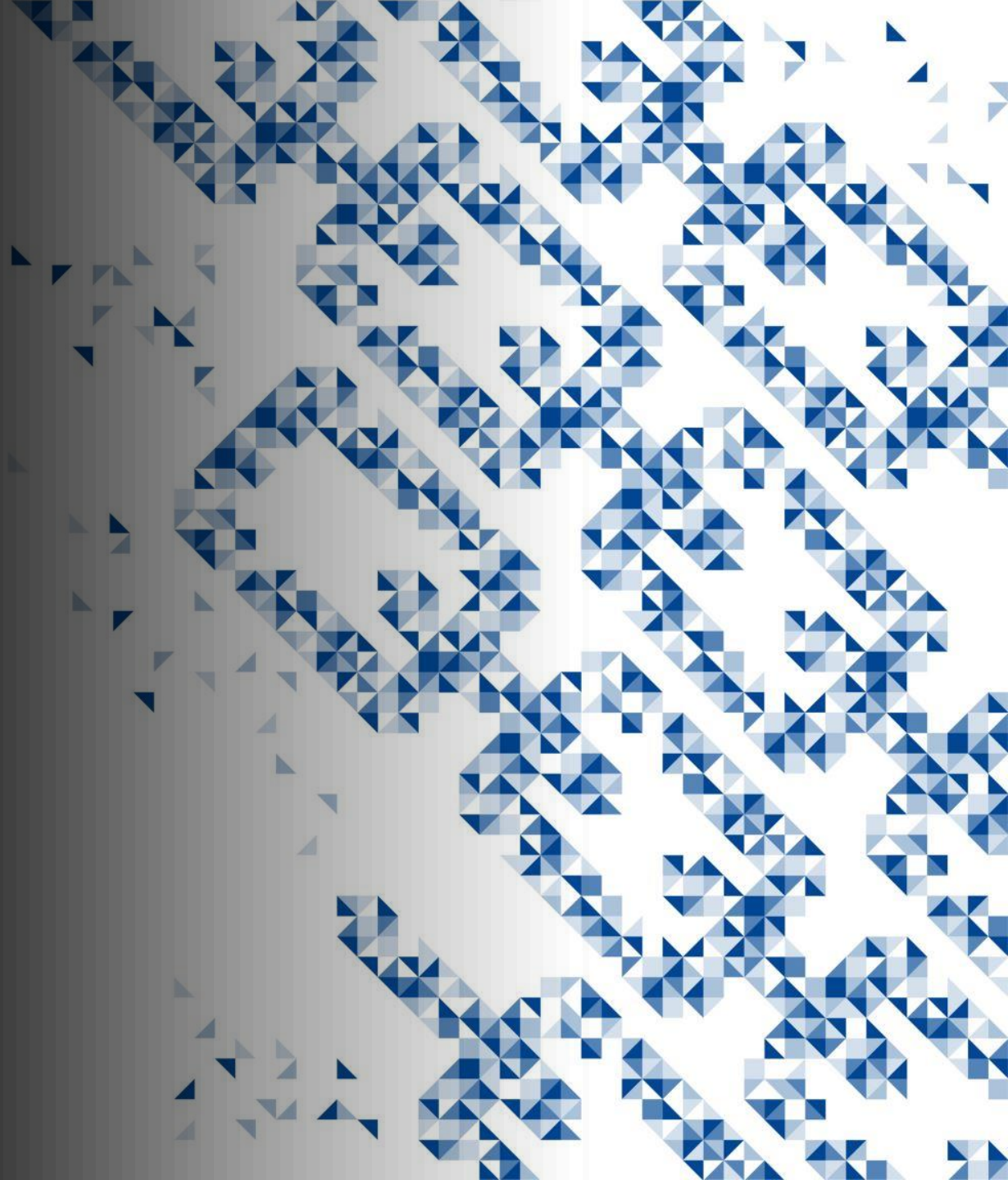

El patrón singleton: una representación en el diagrama UML





Ventajas y desventajas del singleton pattern

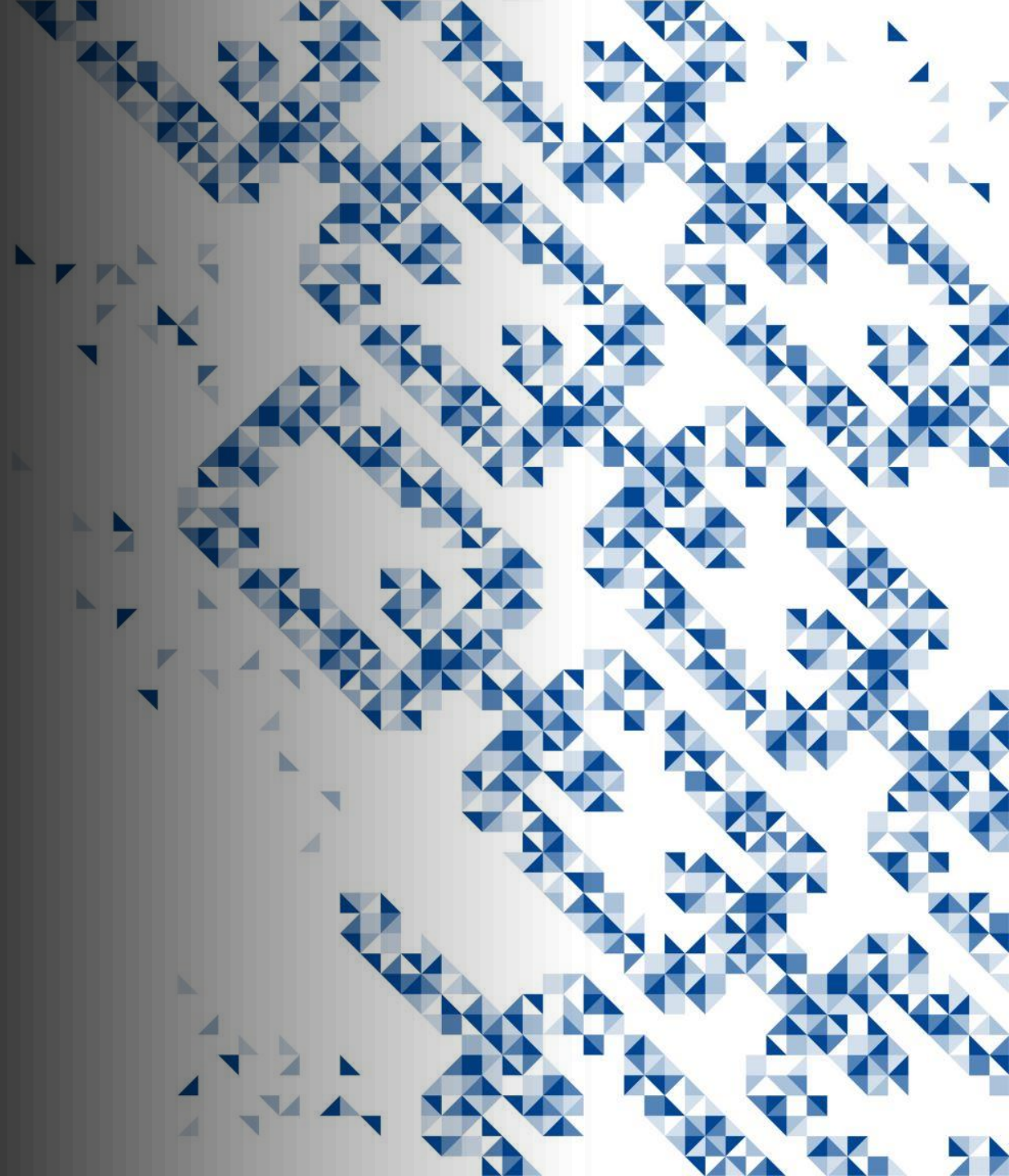
Una visión general de las ventajas

- Al no estar poblado de innumerables variables (globales), un singleton puede escribirse de forma rápida y sencilla. El patrón encapsula su creación, lo que significa que también puede ejercer un control preciso sobre cuándo y cómo se accede a él. Un patrón singleton existente puede derivarse mediante subclases para cumplir nuevas funcionalidades. La funcionalidad que se utiliza se decide dinámicamente. Y, por último, pero no menos importante, un singleton se crea exactamente cuándo se necesita, una característica que se denomina lazy loading. El proceso de instanciar un singleton anteriormente, es decir antes de que se necesite, por otro lado, se llama carga ansiosa.
- 



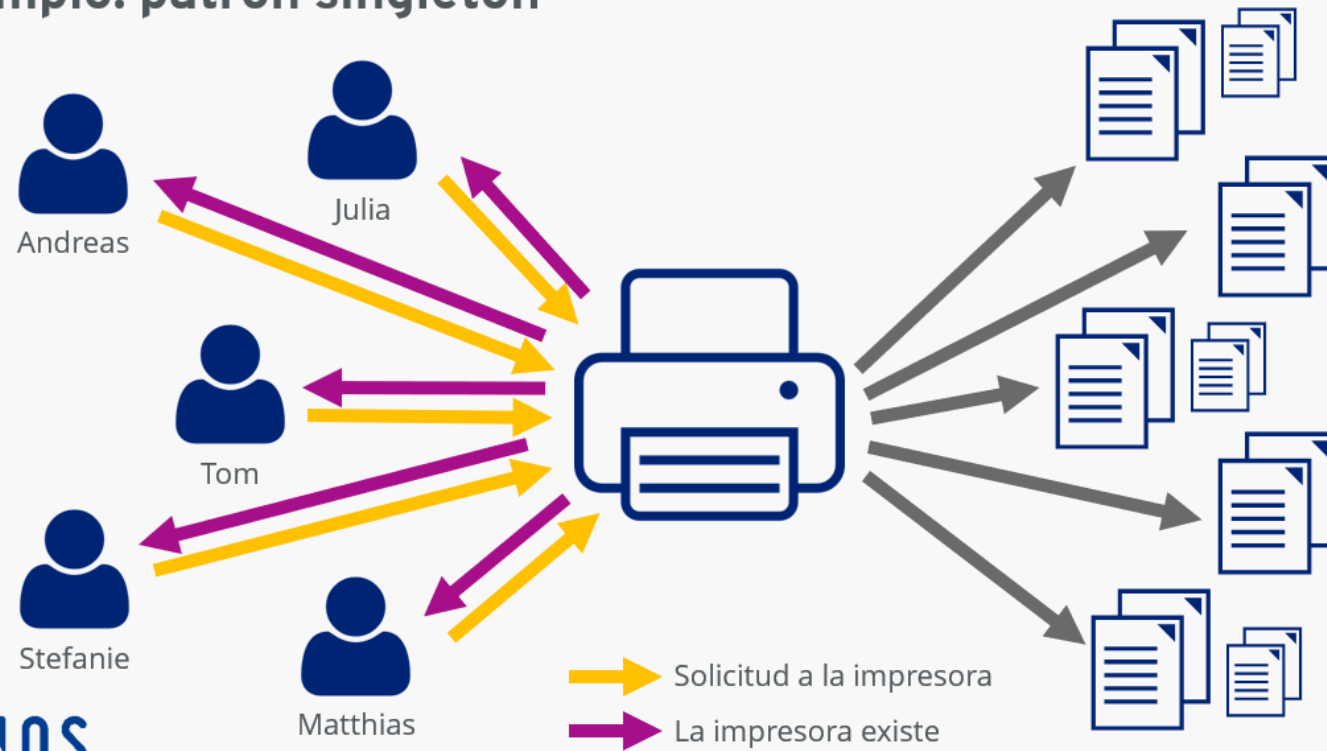
Una visión general de las desventajas

- El uso desinhibido de los singletons conduce a un estado similar al de la programación procedimental (es decir, el no orientado a objetos), y puede llevar a ensuciar el código fuente. La disponibilidad global de patrones singleton plantea riesgos si se manejan datos sensibles. Esto porque si se hacen cambios en el singleton, no se podrá rastrear qué partes del programa están afectadas. Esto dificulta el mantenimiento de software, porque los fallos de funcionamiento son difíciles de rastrear. La disponibilidad global del patrón también dificulta la eliminación de los singleton, ya que los componentes del software siempre pueden referirse a este singleton. En aplicaciones con muchos usuarios (aplicaciones multiusuario), un patrón singleton puede reducir el rendimiento del programa, porque representa un atasco de datos, al ser singular.



ejemplo

Ejemplo: patrón singleton



Patrón builder

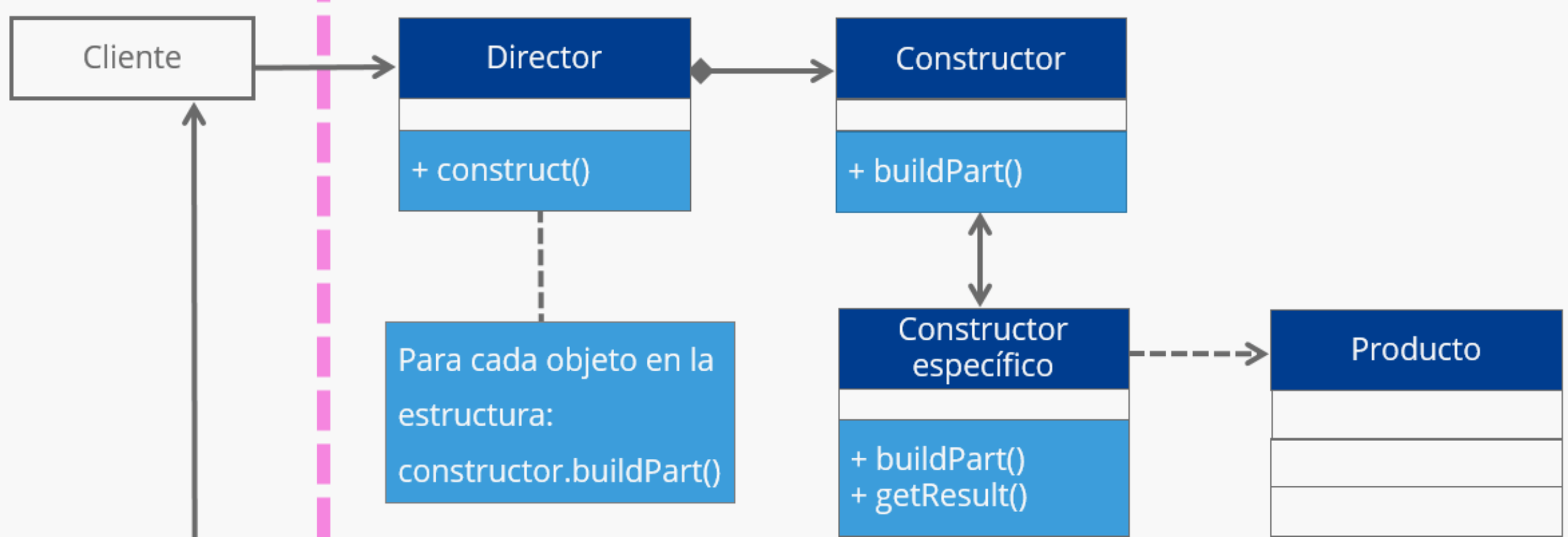
- El Builder Pattern es un tipo de plantilla de patrón de diseño que sirve para resolver tareas de programación en una programación orientada a objetos. Los patrones Builder (o Constructor) facilitan a los desarrolladores el proceso de programación porque no han de rediseñar cada paso que se repite como una rutina de programa.



Un patrón de diseño Builder distingue entre cuatro actores:

- **Director:** este actor construye el objeto complejo con la interfaz del constructor. Es consciente de los requisitos de secuencia del constructor. Con el director, se desvincula la construcción del objeto del cliente.
- **Builder:** ofrece una interfaz para crear los componentes de un objeto (o producto) complejo.
- **Specific builder:** crea las partes del objeto complejo, define (y gestiona) la representación del object, y mantiene la interfaz de salida del objeto.
- **Producto:** es el resultado de la “actividad” del Builder Pattern, es decir, el objeto que se construye.

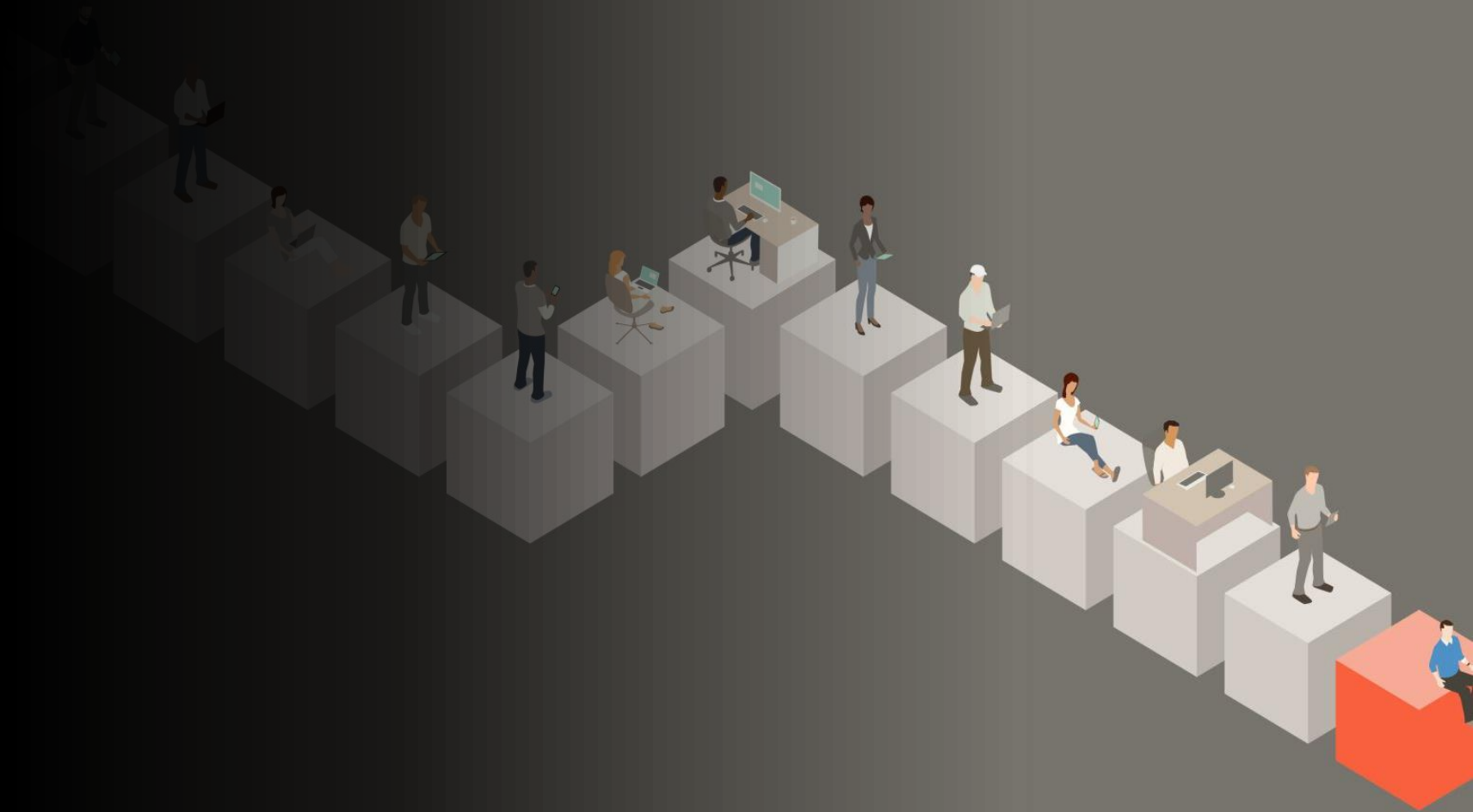




El Builder Pattern como diagrama UML

Ventajas del patrón de diseño Builder

- La construcción y la representación (salida) se incorporan por separado. Las representaciones internas del constructor están ocultas para el director. Las nuevas representaciones como tal pueden integrarse fácilmente utilizando clases de constructores concretos. El proceso de construcción lo controla explícitamente el director. Si hay que hacer cambios, pueden hacerse sin consultar al cliente.



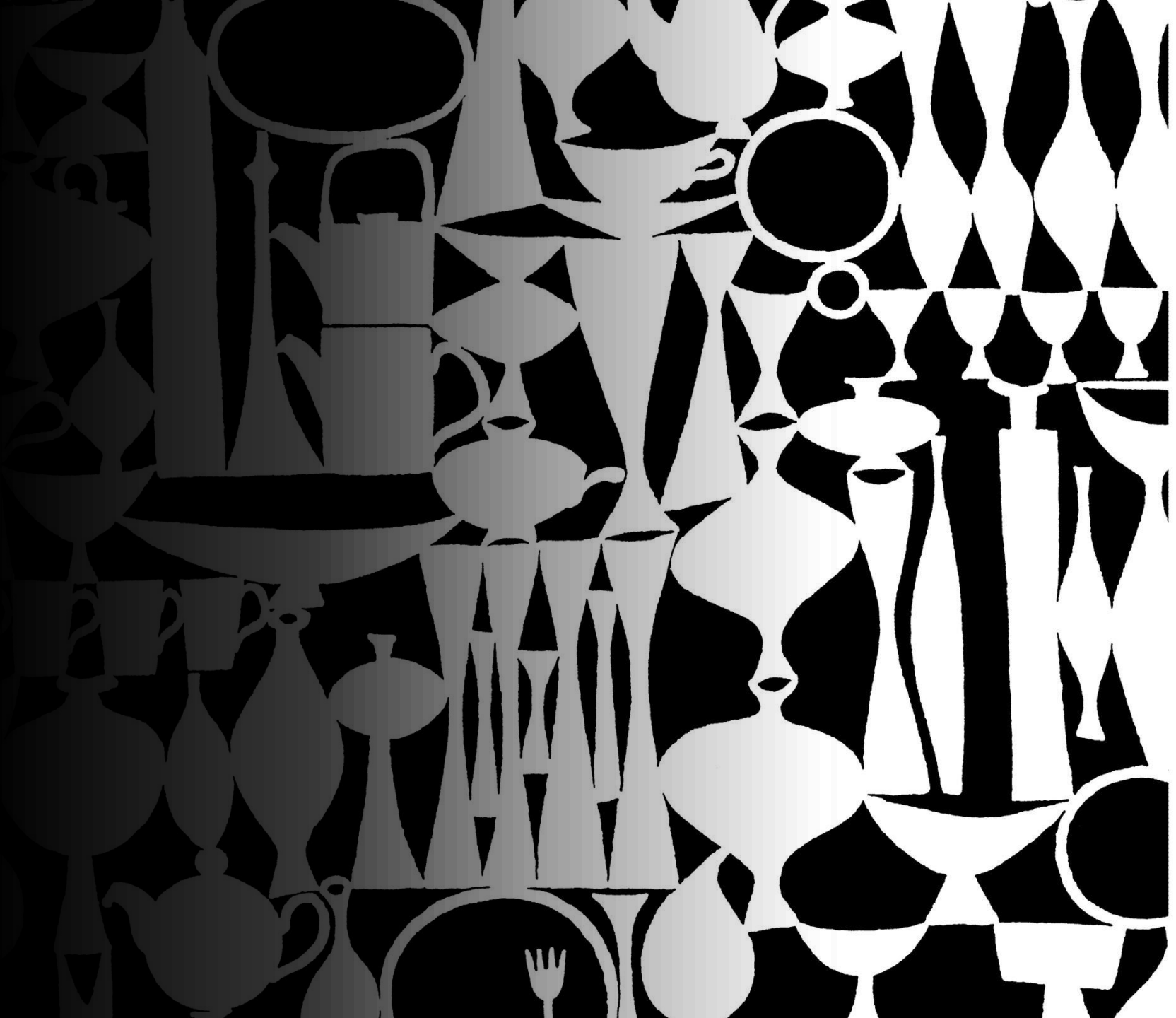
A 3D puzzle with one red piece standing out. The puzzle is composed of white and grey pieces, with one red piece in the center. The red piece is slightly raised and has a glossy finish. The background is a gradient from dark grey to light grey.

Inconvenientes de Builder

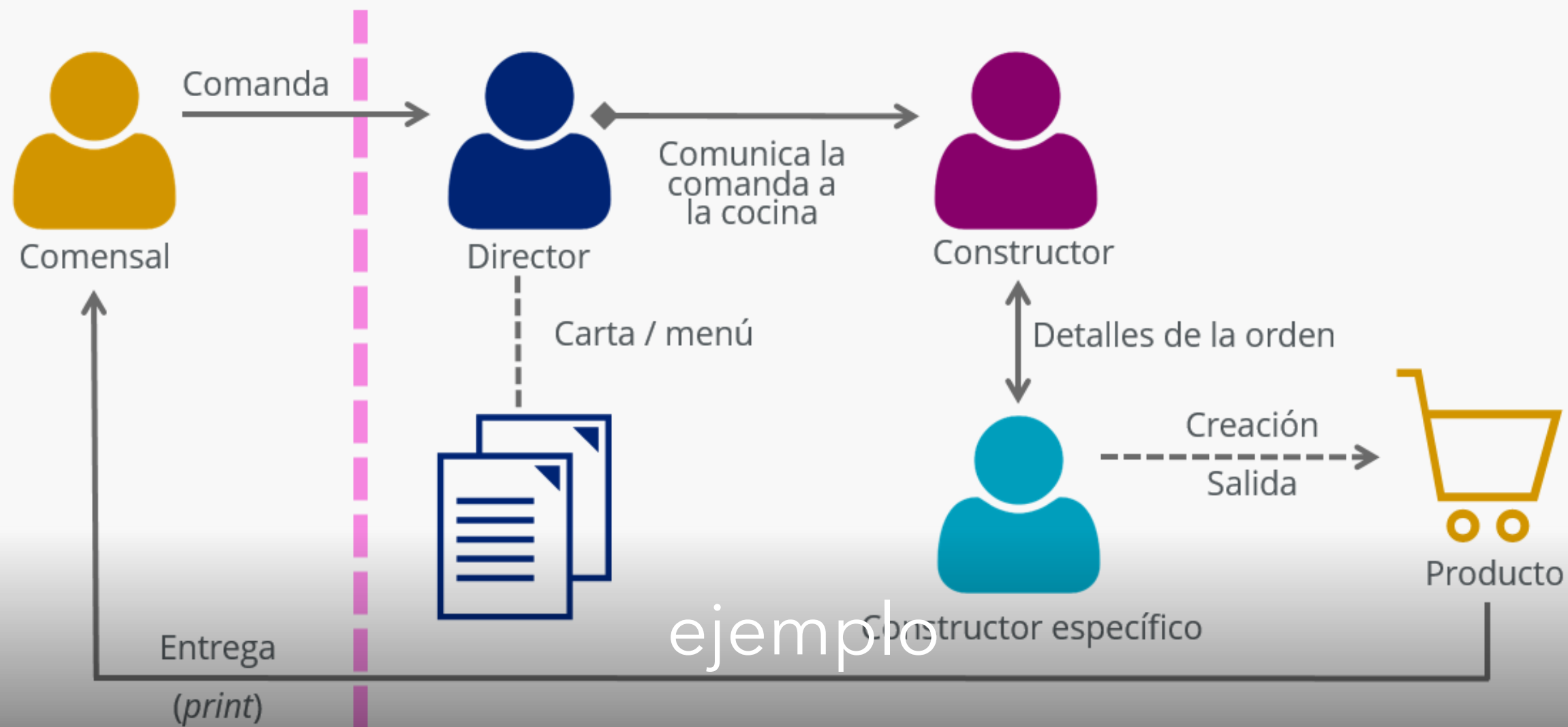
- El patrón Constructor consta de un **fuerte vínculo** entre el producto, el constructor específico y las clases del proceso de diseño, así que puede ser **difícil hacer cambios en el proceso básico**. La construcción de los objetos requiere conocer su uso y su entorno concretos. Utilizar patrones conocidos, como el patrón de diseño Builder, puede hacer que los programadores pasen por alto soluciones más sencillas y elegantes. En el fondo, muchos desarrolladores consideran que este es uno de los patrones de diseño menos importantes.

¿Cuándo se utiliza el Builder Pattern?

- Una manera de ilustrar el patrón de diseño Builder es considerando el ejemplo de un restaurante al que un cliente realiza un pedido. Una vez que han recibido el pedido, los cocineros actúan para elaborarlo. Todo el proceso hasta la entrega como tal se hace “entre bambalinas”; el cliente no ve lo que ocurre en la cocina y solo recibe el resultado (el *print* en lenguaje de programación).



La mecánica de Builder Pattern con un ejemplo



builder

```
1. package com.refactorizando.patterns.builder;
2.
3. public interface Builder {
4.
5.     void setCarType(CarType type);
6.     void setSeats(int seats);
7.     void setEngine(Engine engine);
8.     void setWheels(int wheels);
9.     void setLeatherSeats(Boolean leatherSeats);
10.    void setBatteries(int batteries);
11.    void setConvertible(Boolean convertible);
12.    void setElectricCar(Boolean electricCar);
13. }
```

product

```
1. package com.refactorizando.patterns.builder;
2.
3. public class Car {
4.
5.     private CarType carType;
6.     private Integer seats;
7.     private Integer bigWheels;
8.     private Engine engine;
9.     private Boolean leatherSeats;
10.    private Integer batteries;
11.    private Boolean convertible;
12.    private Boolean electricCar;
13.
14.    public Car(CarType carType, Integer seats, Integer bigWheels,
15.              Engine engine, Boolean leatherSeats, Integer batteries, Boolean convertible,
16.              Boolean electricCar) {
17.        this.carType = carType;
18.        this.seats = seats;
19.        this.bigWheels = bigWheels;
20.        this.engine = engine;
21.        this.leatherSeats = leatherSeats;
22.        this.batteries = batteries;
23.        this.convertible = convertible;
24.        this.electricCar = electricCar;
25.    }
26.
27.    public void setBigWheels(Integer bigWheels) {
28.        this.bigWheels = bigWheels;
29.    }
30.
31.    public void setEngine(Engine engine) {
32.        this.engine = engine;
33.    }
34.
35.    public void setLeatherSeats(Boolean leatherSeats) {
36.        this.leatherSeats = leatherSeats;
37.    }
```

```
1. package com.refactorizando.patterns.builder;
2.
3. public class Engine {
4.
5.     private Integer engineCapacity;
6.
7.     public Integer getEngineCapacity() {
8.         return engineCapacity;
9.     }
10.
11.    public void setEngineCapacity(Integer engineCapacity) {
12.        this.engineCapacity = engineCapacity;
13.    }
14.
15.    public Engine(Integer engineCapacity) {
16.        this.engineCapacity = engineCapacity;
17.    }
18. }
```

y como nos encontramos en un concesionario de venta de coches, necesitamos indicar el tipo de nuestro coche:

```
1. package com.refactorizando.patterns.builder;
2.
3. public enum CarType {
4.     LUXURY, SPORT, BERLINA, SMALL
5. }
```


Specific builder

```
public class CarBuilder implements Builder {  
  
    private Integer seats;  
    private CarType carType;  
    private Integer bigWheels;  
    private Engine engine;  
    private Boolean leatherSeats;  
    private Integer batteries;  
    private Boolean convertible;  
    private Boolean electricCar;  
  
    @Override  
    public void setCarType(CarType type) {  
        this.carType = type;  
    }  
  
    @Override  
    public void setSeats(int seats) {  
        this.seats = seats;  
    }  
  
    @Override  
    public void setEngine(Engine engine) {  
        this.engine = engine;  
    }  
  
    @Override  
    public void setWheels(int wheels) {  
        this.bigWheels = wheels;  
    }  
}
```



Director

```
. package com.refactorizando.patterns.builder;
.
. public class Dealership {
.
.     public void createLuxuryCar(Builder builder) {
.         builder.setCarType(CarType.LUXURY);
.
.         Engine engine = new Engine(2700);
.
.         builder.setSeats(4);
.         builder.setElectricCar(Boolean.TRUE);
.         builder.setBatteries(2);
.         builder.setConvertible(Boolean.TRUE);
.         builder.setLeatherSeats(Boolean.TRUE);
.         builder.setEngine(engine);
.
.     }
.
.     public void createSmallCar(Builder builder) {
.         builder.setCarType(CarType.SMALL);
.
.         Engine engine = new Engine(1200);
.         builder.setBatteries(1);
.         builder.setEngine(engine);
.         builder.setSeats(4);
.
.     }
.
.     public void createBerlinaCar(Builder builder) {
.         builder.setCarType(CarType.BERLINA);
.
.         Engine engine = new Engine(1900);
.         builder.setSeats(5);
.         builder.setElectricCar(Boolean.TRUE);
.         builder.setBatteries(2);
.         builder.setEngine(engine);
.
.     }
```

Ejemplo de uso practico

```
1. package com.refactorizando.patterns.builder;
2.
3. public class Application {
4.
5.     public static void main(String[] args) {
6.
7.         Dealership dealership = new Dealership();
8.
9.         // The client wants a simple and small car
10.        CarBuilder builder = new CarBuilder();
11.        dealership.createSmallCar(builder);
12.
13.        Car car = builder.getCar();
14.        System.out.println("My : " + car.getCarType() + " car");
15.
16.    }
17.
18. }
```