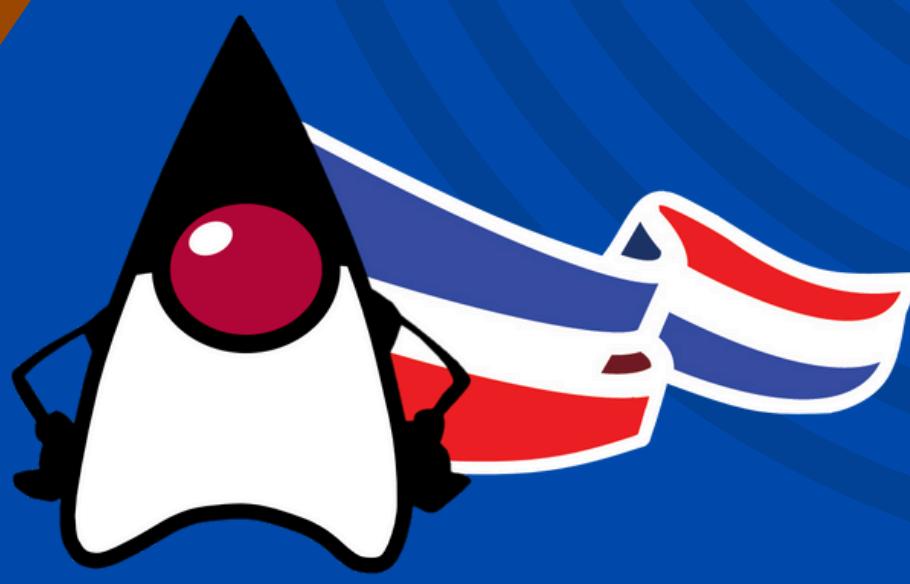
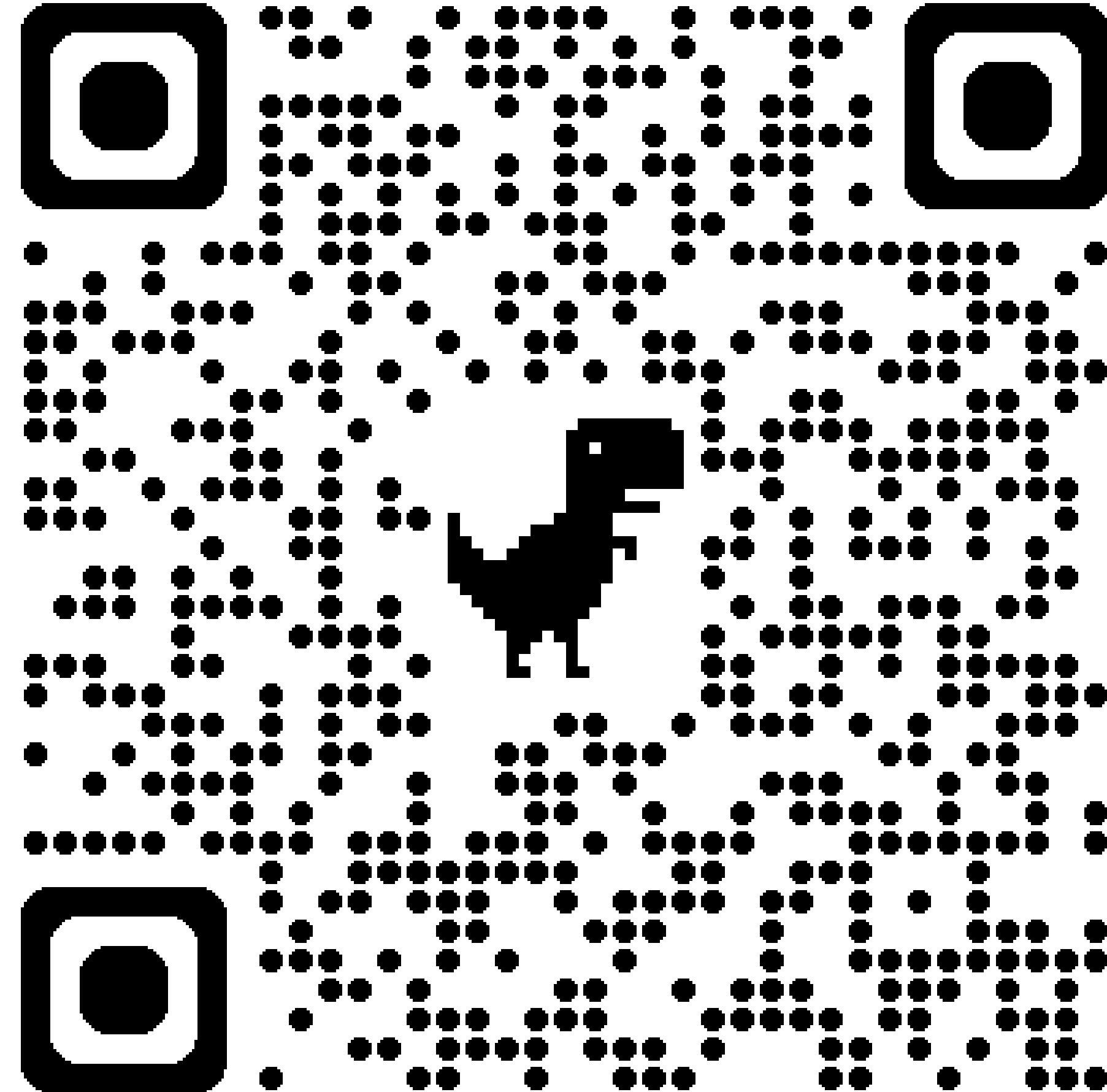




**Piensa en Patrones**

Diseñando Software como un Arquitecto





# ¿Por Qué Estamos Aquí?

**Para aprender a pensar como arquitectos de software, no solo como programadores.**

Los patrones de diseño nos permiten ir más allá del “código que funciona” y empezar a construir soluciones que son **reutilizables, escalables y mantenibles.**

**Para hablar un lenguaje común entre desarrolladores.**

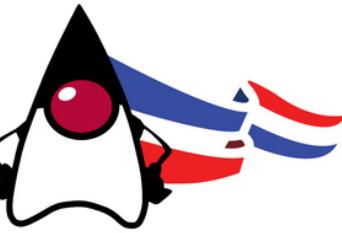
UML y los patrones de diseño nos ofrecen una vocabulario universal para comunicar ideas, evitando malentendidos y reduciendo la fricción en los equipos.

**Para enfrentar problemas reales con soluciones probadas.**

Los patrones no son teoría lejana: nacieron de la experiencia de miles de proyectos. Aquí veremos cómo aplicarlos en escenarios concretos y reconocer cuándo usarlos y cuándo no.

**Para mejorar la calidad y el futuro de nuestro software.**

Aplicar patrones correctamente significa menos deuda técnica, más claridad en el código y una base más sólida para evolucionar nuestros sistemas con el tiempo.



# Agenda

- ① Día 1 -- Introducción a UML + Patrones Creacionales
- ② Día 2 -- Patrones Estructurales
- ③ Día 3 -- Patrones de Comportamiento



# Enfocado en propósito y futuro

Hoy no estamos aquí solo para aprender a dibujar diagramas o memorizar patrones. UML es mucho más que un conjunto de símbolos: es una manera de **pensar, comunicar y construir software** que trasciende el código.

Los patrones de diseño, por su parte, son el puente entre la teoría y la práctica, entre la experiencia de quienes vinieron antes y los retos que nos esperan. Lo que hoy parece un lenguaje abstracto, mañana será la base que te permitirá **explicar tus ideas con claridad, resolver problemas complejos y diseñar sistemas preparados para el futuro**.



# Orador



- Co-Founder & CEO at Alphnology
- Co-Founder & VP of Engineering at ClimaCall
- Vaadin Champion
- JUG Leader (Java Dominicana)
- JConf Dominicana Staff
- Java Expert
- Software Architect
- Speaker
- Lecturer
- Consultant

# Descargo de responsabilidad

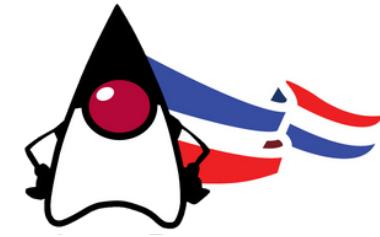
- La información proporcionada en esta presentación tiene como objetivo educativo y se basa en experiencia personal y conocimientos actuales. Si bien se ha hecho todo lo posible para garantizar la precisión y la actualidad de la información presentada, no se puede garantizar su exactitud completa.
- El uso de las tecnologías mencionadas, está sujeto a los términos y condiciones de cada herramienta. Es responsabilidad del usuario realizar su propia investigación y cumplir con las directrices y políticas de cada tecnología antes de implementarlas en su entorno de producción.
- Además, cabe señalar que las mejores prácticas y las soluciones presentadas en esta presentación pueden variar según los requisitos y las circunstancias específicas de cada proyecto. Recomiendo encarecidamente realizar pruebas exhaustivas y consultar con profesionales capacitados antes de implementar cualquier solución en un entorno de producción.
- En resumen, mientras que esta presentación busca proporcionar información útil y práctica, el uso de las tecnologías y las decisiones de implementación son responsabilidad del usuario final. No se asume ninguna responsabilidad por los resultados derivados de la aplicación de los conceptos discutidos en esta presentación.

# ¿Qué es UML?

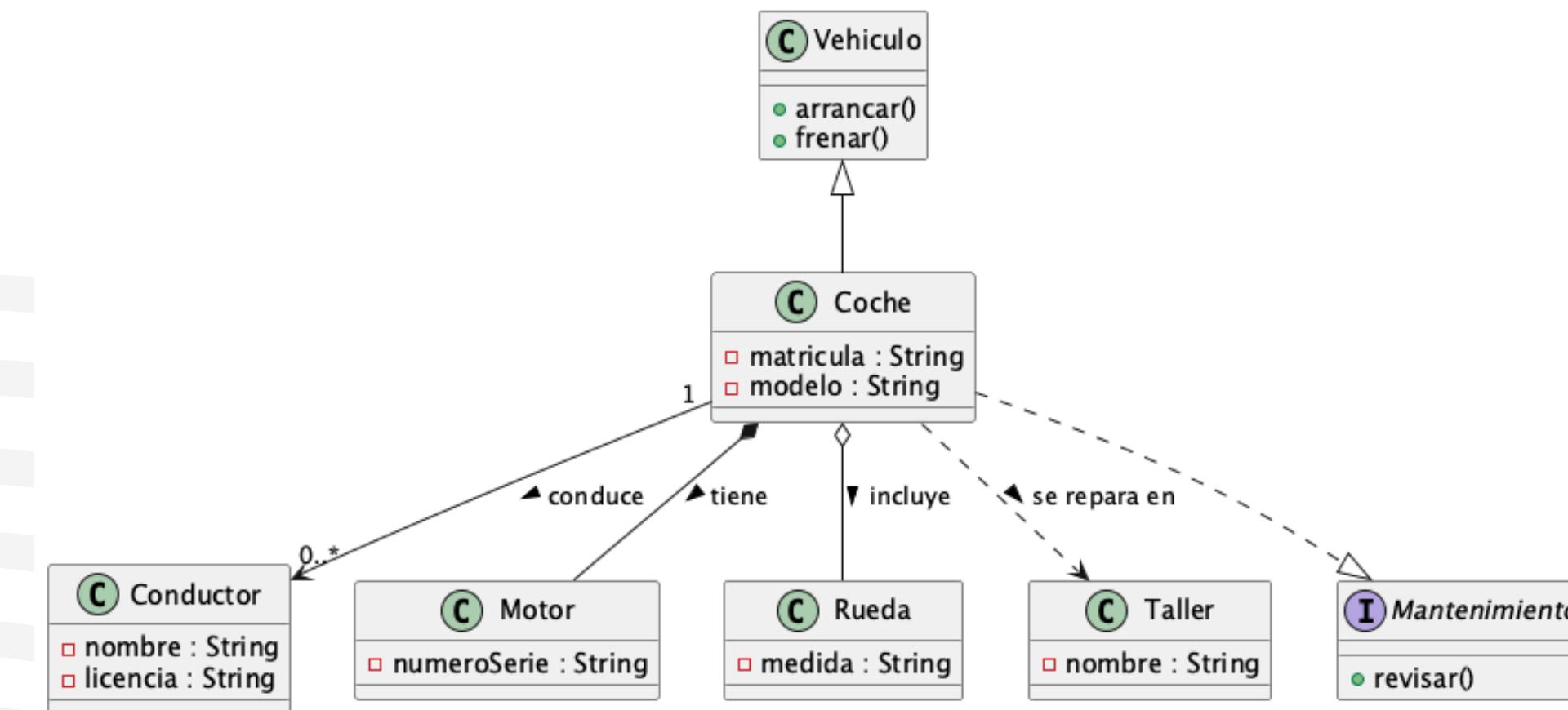
- **UML** (Unified Modeling Language / Lenguaje Unificado de Modelado) es un lenguaje gráfico estándar para **visualizar, especificar, construir y documentar** sistemas de software, incluyendo su estructura y comportamiento.
- No es un lenguaje de programación; es un **lenguaje de modelado**: describe qué hará el sistema, cómo se estructura, cuáles son las interacciones, etc.
- Algunos usos importantes:
  - Como **herramienta de comunicación** entre arquitectos, desarrolladores, y otros stakeholders.
  - Para **documentación** del sistema.
  - Para **diseño previo al código**, especialmente útil si el proyecto es grande o colaborativo.
  - Para validar ideas de diseño antes de implementar.

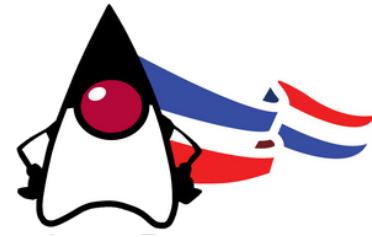


# Diagrama de clases

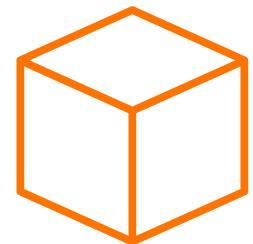


- Es uno de los **diagramas estructurales** más usados en UML. Muestra clases, atributos, operaciones (métodos) y relaciones estáticas entre clases.
- Permite entender la estructura del sistema: qué clases existen, cómo se relacionan, qué responsabilidades tienen.
- Una clase se representa típicamente con un rectángulo dividido en tres partes:
  - Nombre de la clase (en la parte superior).
  - Atributos (parte del medio).
  - Operaciones / Métodos (parte inferior).
- Se puede mostrar visibilidad de atributos/métodos: Público (+), Privado (-), Protegido (#), Paquete (~), Derivado (/), Estático (**subrayado**).
- Hay dos alcances para los miembros (**clasificadores e instancias**) Los clasificadores son miembros estáticos, mientras que las instancias son las instancias específicas de la clase. Si estás familiarizado con POO, esto no es nada nuevo.





# Componentes del diagrama de clases



**Nombre de la clase:** claro, único dentro del contexto del modelo.



**Atributos:** con tipo, visibilidad y (opcionalmente) multiplicidad, valores por defecto.



**Métodos / operaciones:** nombre, visibilidad, parámetros, tipo de retorno.



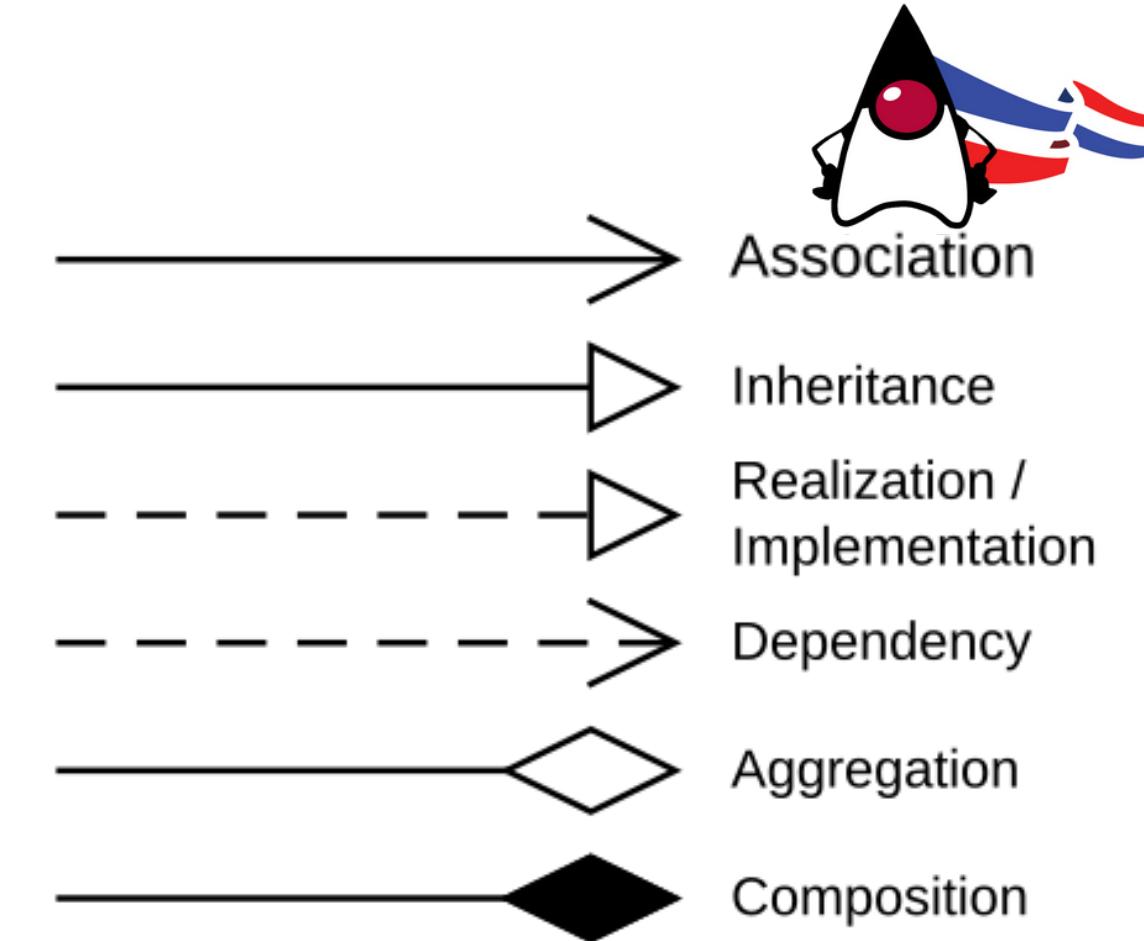
**Interfaces:** similares a clases, pero sólo con firmas de métodos que luego concretan otras clases.



**Relaciones:** este punto lo vamos a detallar, porque es clave para modelar bien.

# Relaciones entre clases

- Hay seis tipos principales de relaciones entre clases: , **Asociación, Generalización/herencia, Realización/Implementación, Dependencia, Agregación y Composición.**
- Una relación es un término general que abarca los tipos específicos de conexiones lógicas presentes en los diagramas de clases y objetos. UML agrupa las siguientes relaciones:
- **Relaciones a nivel de instancia:**
  - Dependencia, Asociación, Agregación y Composición
- **Relaciones a nivel de clase**
  - Generalización/herencia, Realización/Implementación
- **Multiplicidad:** Esta relación de asociación indica que (al menos) una de las dos clases relacionadas hace referencia a la otra. Esta relación suele describirse como "A tiene una B" (una gata tiene gatitos, y los gatitos tienen una gata).



Flechas para las relaciones

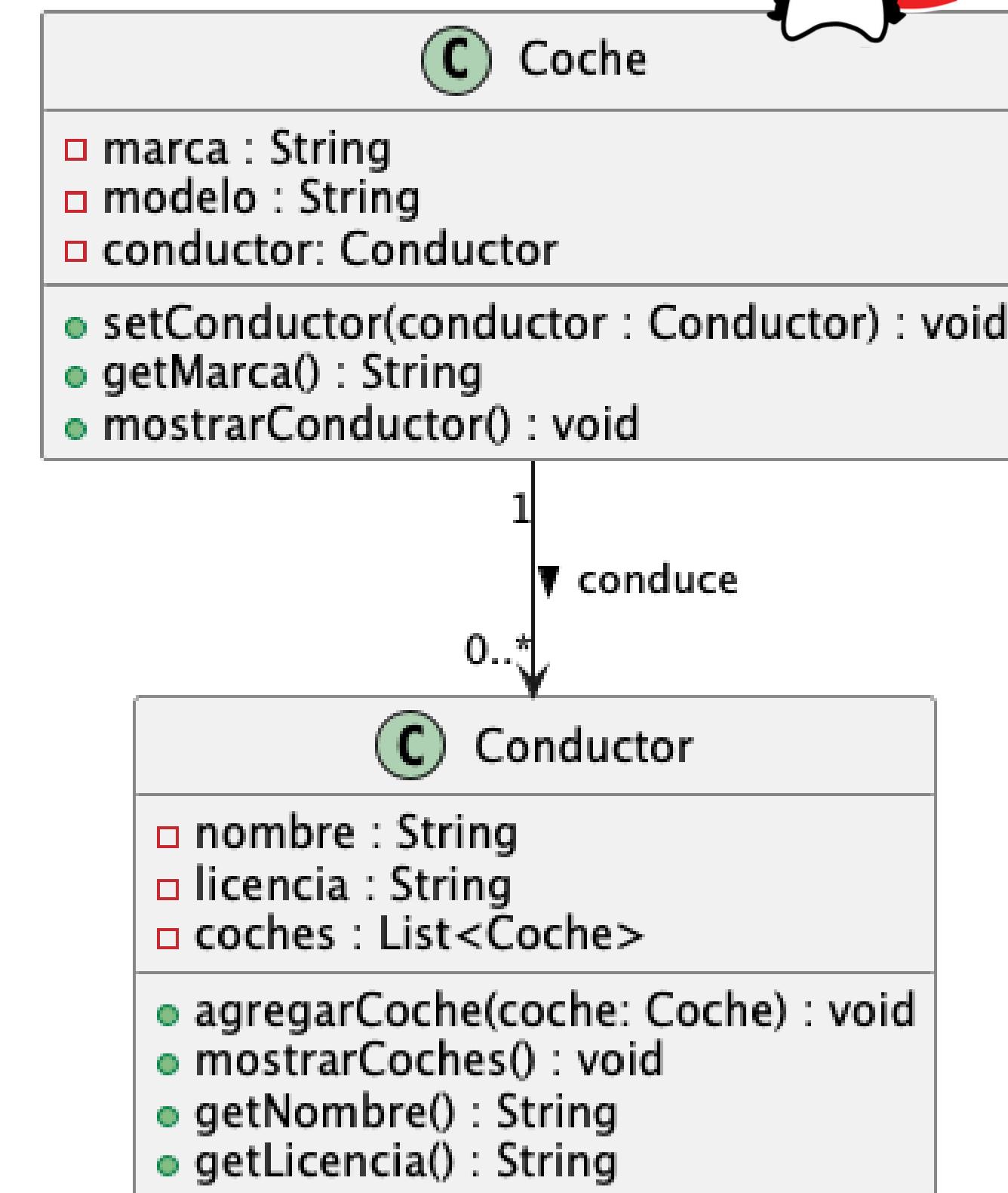
<b>0</b>	No hay casos (raros)
<b>0..1</b>	Ninguna instancia, o una sola instancia
<b>1</b>	Exactamente un caso
<b>1..1</b>	Exactamente un caso
<b>0..*</b>	Cero o más instancias
<b>*</b>	Cero o más instancias
<b>1..*</b>	Una o más instancias



# Tipos de relaciones entre clases

## Asociación

- Indica que **una propiedad de una clase contiene una referencia a una instancia (o instancias) de otra clase**.
  - La asociación es la relación **más utilizada** entre una clase y otra clase, lo que significa que existe una conexión entre un tipo de objeto y otro tipo de objeto.
- Las combinaciones y agregaciones también pertenecen a las relaciones asociativas**, pero las relaciones entre clases de afiliaciones son más débiles que las otras dos.
- Hay cuatro tipos de asociaciones: **asociaciones bidireccionales, asociaciones unidireccionales, autoasociación y asociaciones de números múltiples**.
  - Por ejemplo: coches y conductores, un coche corresponde a un conductor en particular y un conductor puede conducir varios coches.



→ Association

→ Inheritance

→ Implementation

→ Dependency

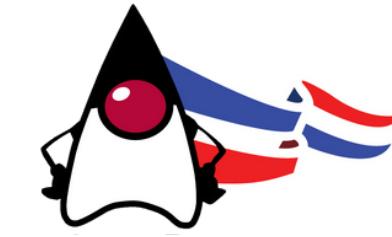
→ Aggregation

→ Composition

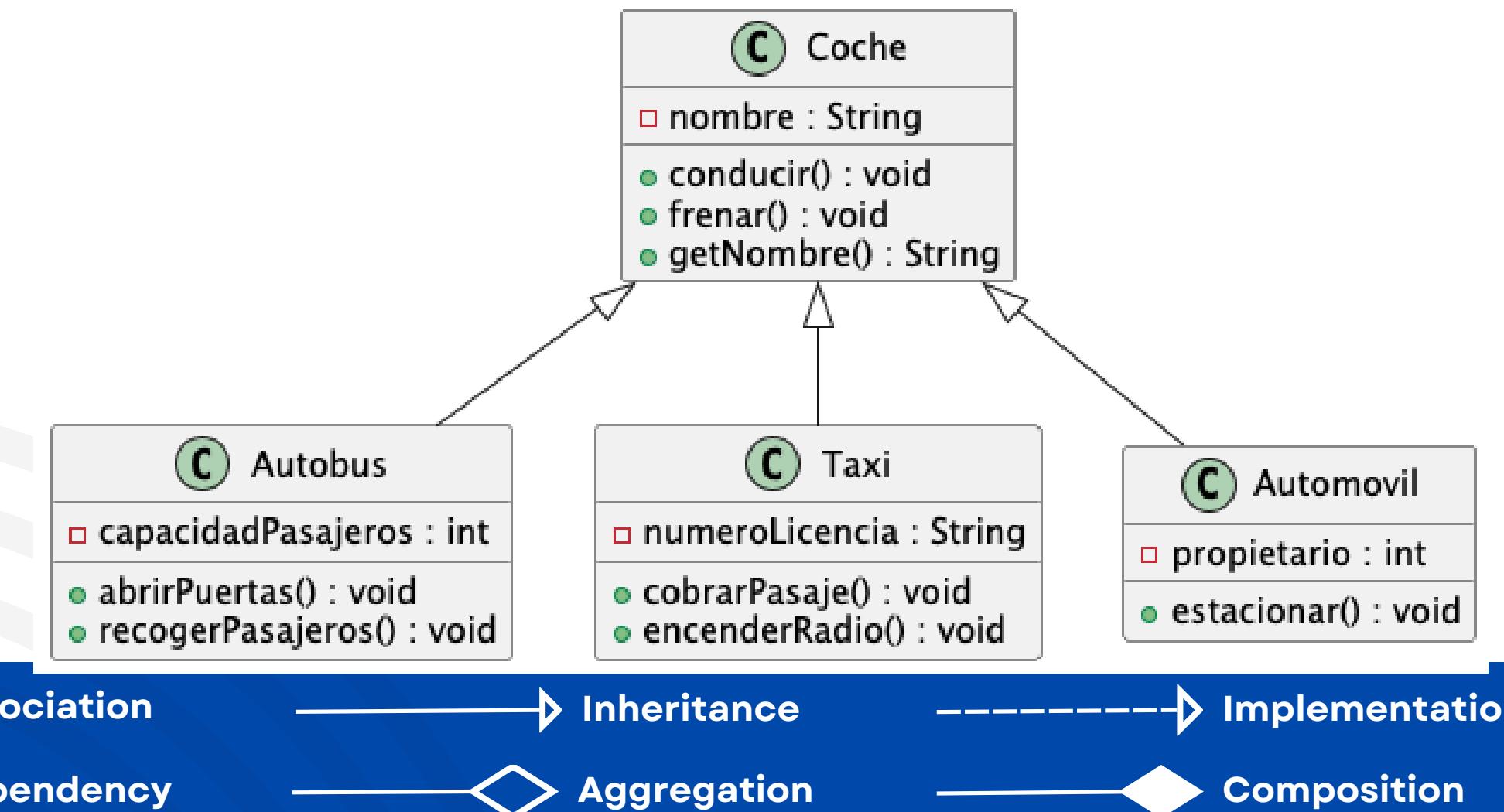


# Tipos de relaciones entre clases

## Generalización/herencia



- La herencia también se denomina **generalización** y se utiliza para describir la relación entre las clases padre e hijo. Una clase principal también se denomina clase base y una subclase también se denomina clase derivada.
- En la relación de herencia, la subclase hereda todas las funciones de la clase principal y la clase principal tiene todos los atributos, métodos y subclases. Las subclases contienen información adicional además de la misma información que la clase principal.
- Por ejemplo: autobuses, taxis y automóviles son coche, todos tienen nombres y todos pueden estar en la carretera.

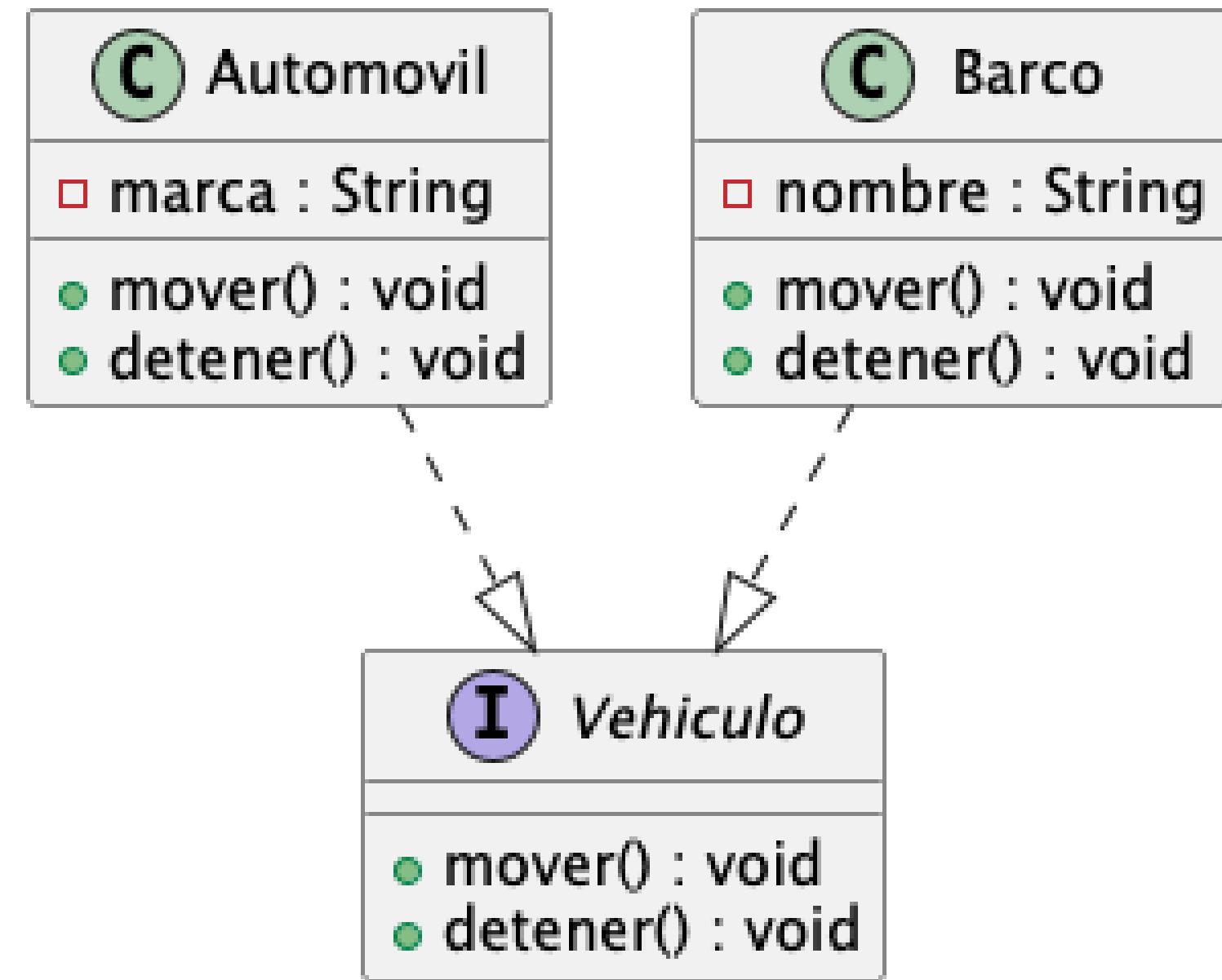




# Tipos de relaciones entre clases

## Realización/Implementación

- La implementación se utiliza principalmente para especificar **la relación entre las interfaces y las clases de implementación**.
- **Una interfaz** (incluida una **clase abstracta**) es una colección de métodos. En una relación de implementación, una clase implementa una interfaz y los métodos de la clase implementan todos los métodos de la declaración de la interfaz.
- Por ejemplo: los automóviles y los barcos son vehículos, y el vehículo es solo un concepto abstracto de una herramienta móvil, y el barco y el vehículo realizan las funciones móviles específicas.



→ Association

→ Dependency

→ Inheritance

→ Aggregation

→ Implementation

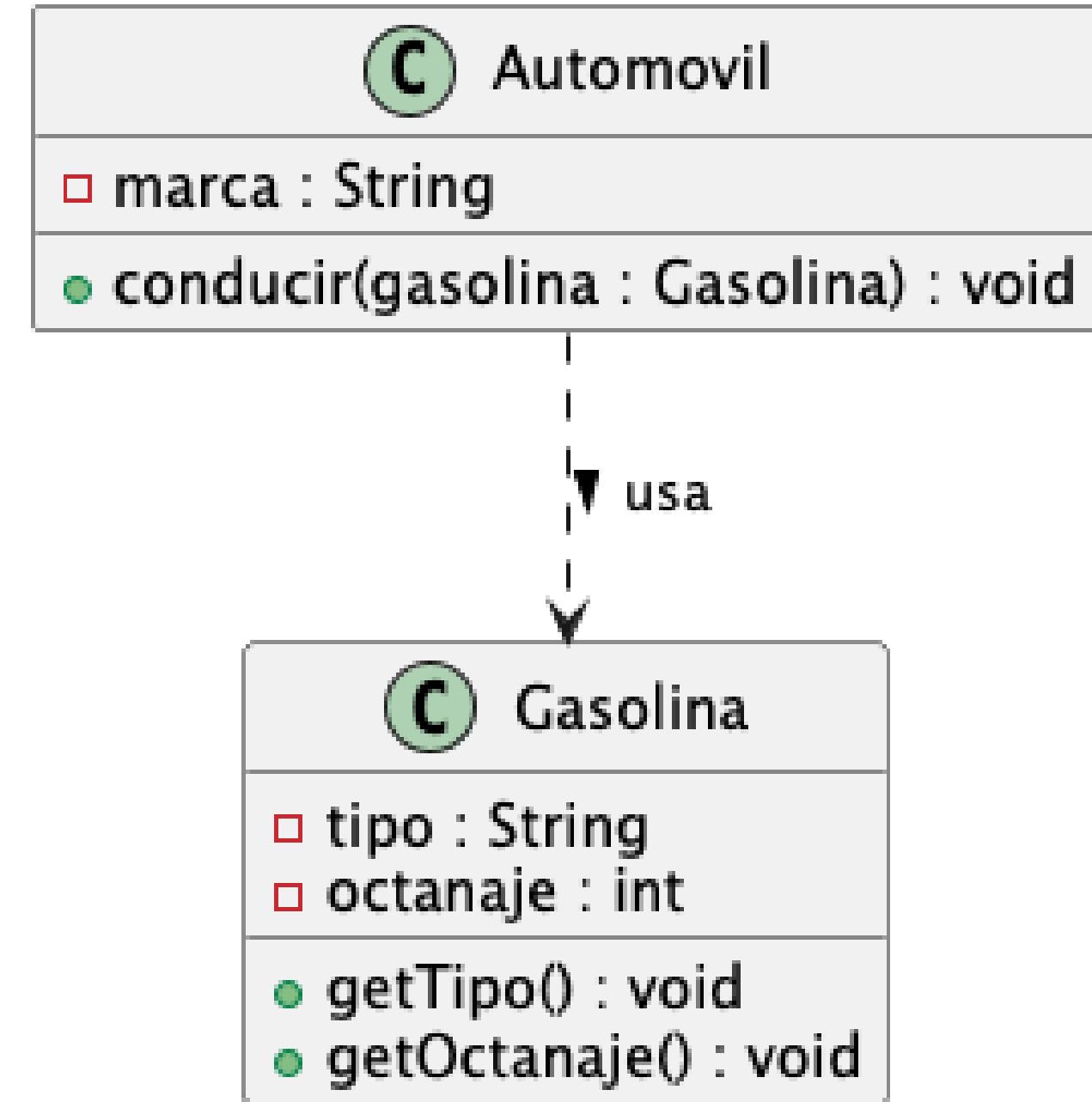
→ Composition



# Tipos de relaciones entre clases

## Dependencia

- Una dependencia es un tipo de asociación donde existe una conexión semántica entre elementos del modelo dependientes e independientes, en la mayoría de los casos, **las dependencias se reflejan en los métodos de una clase que utilizan el objeto de otra clase como parámetro.**
- Una relación de dependencia es una relación de “uso”. Un cambio en una cosa en particular puede afectar a otras cosas que la usan, y usar una dependencia cuando es necesario indicar que una cosa usa otra. Por ejemplo: El auto depende de la gasolina. Si no hay gasolina, el automóvil no podrá conducir.
- Por ejemplo: Un automóvil no tiene gasolina como atributo permanente, sino que usa gasolina en un método (**conducir**). Por lo tanto, si cambia la clase **Gasolina** (atributos, métodos), el método de **Automovil** debe cambiar. Esto ejemplifica una **relación de dependencia**.



→ Association

→ Dependency

→ Inheritance

→ Aggregation

→ Implementation

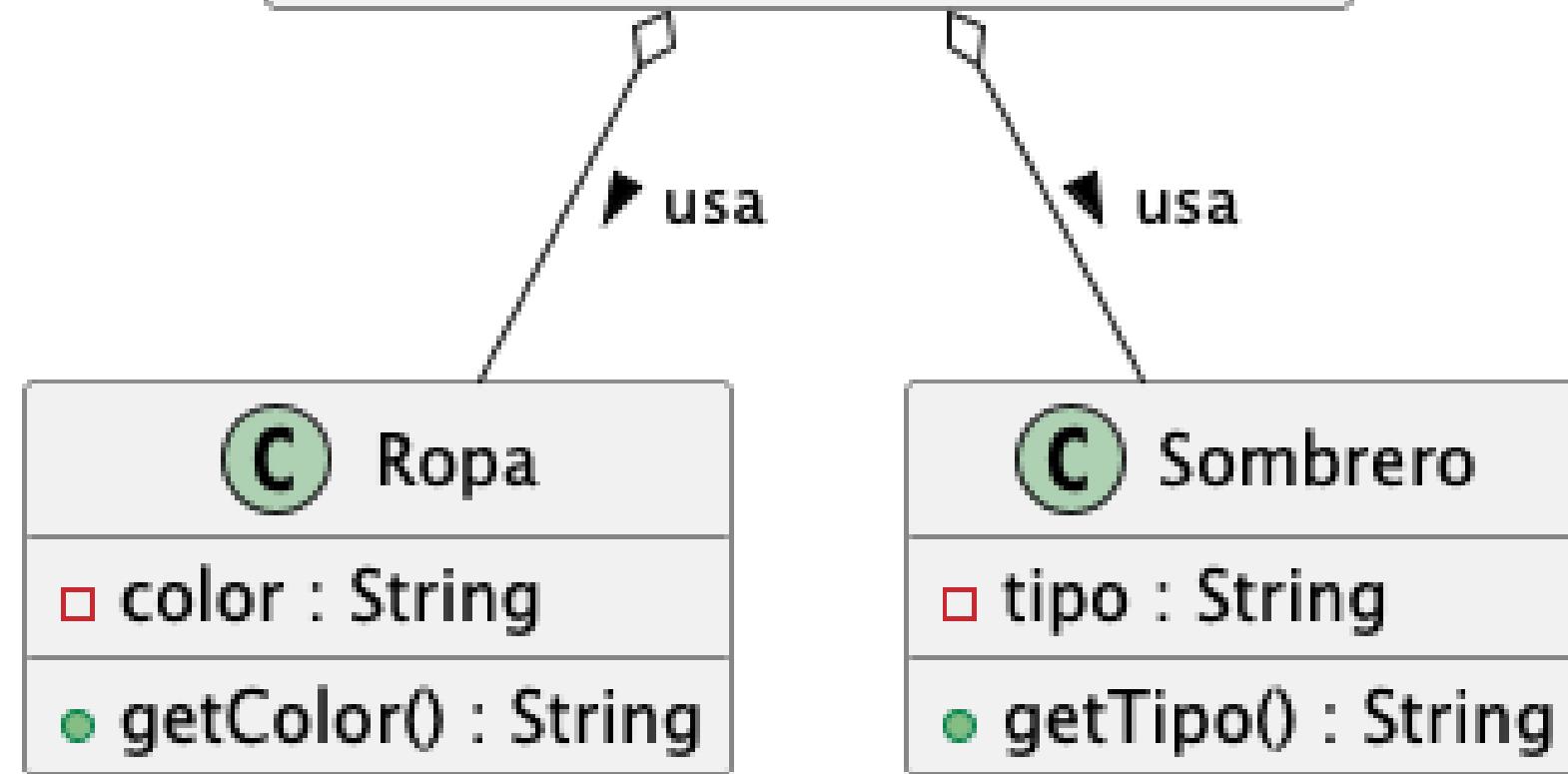
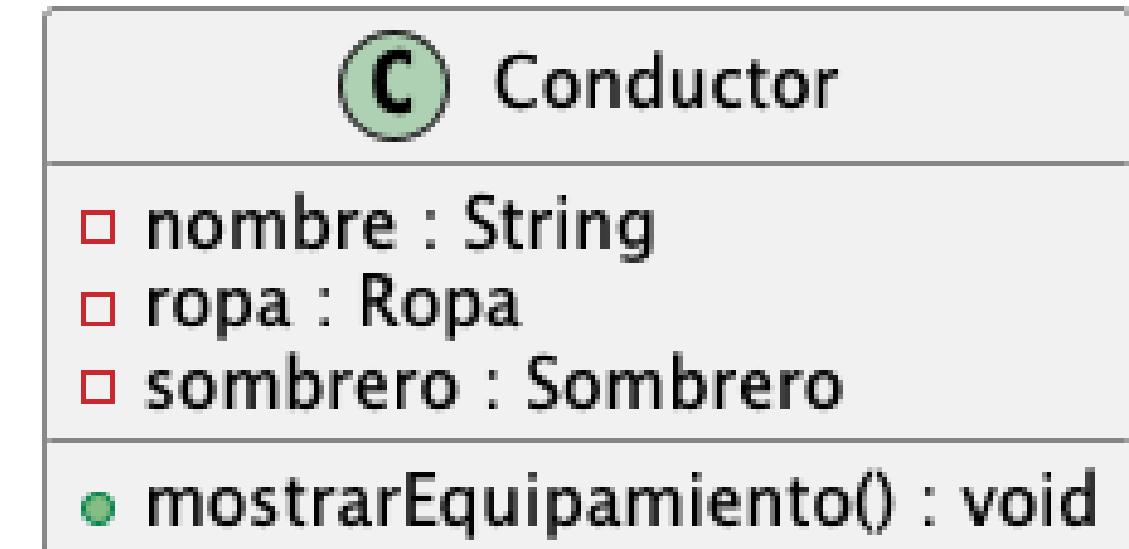
→ Composition



# Tipos de relaciones entre clases

## Agregación

- La agregación es una variante de la relación de asociación, es más específica que la asociación. **La relación entre el todo y la parte, y el todo y la parte se pueden separar.**
- Las relaciones agregadas también representan la relación entre el todo y una parte de la clase, los objetos miembros son parte del objeto general, pero el objeto miembro puede existir independientemente del objeto general.
- Por ejemplo, los conductores de autobús y la ropa y los sombreros de trabajo son parte de la relación general, pero se pueden separar. La ropa de trabajo y los sombreros se pueden usar en otros conductores. Los conductores de autobuses también pueden usar otra ropa de trabajo y sombreros.



→ Association

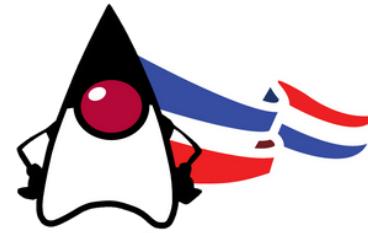
→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

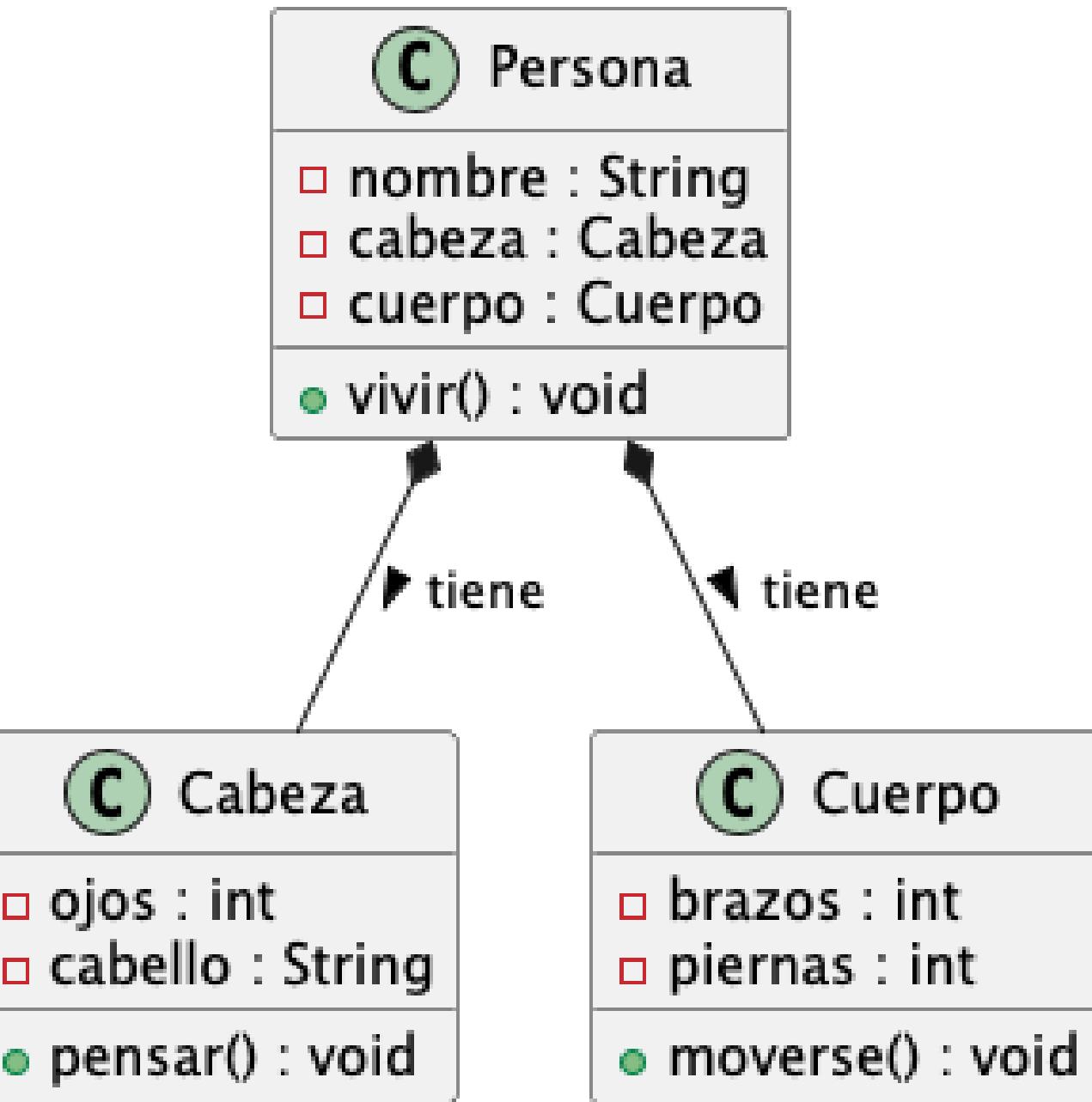




# Tipos de relaciones entre clases

## Composición

- La relación de agregación compuesta (coloquialmente llamada composición) es una forma más robusta de agregación, donde el agregado controla el ciclo de vida de los elementos que agrega (**relación entre el todo y la parte, pero el todo y la parte no se pueden separar**).
- La relación de combinación representa la relación entre el todo y la parte de la clase, y el total y la parte tienen una duración constante. Una vez que el objeto general no existe, algunos de los objetos no existirán y todos morirán en la misma vida. Por ejemplo, una persona está compuesta por una cabeza y un cuerpo. Los dos son inseparables y coexisten.
- Por ejemplo: Una Persona está compuesta por una Cabeza y un Cuerpo.
  - Persona (todo): controla el ciclo de vida de sus partes.
  - Cabeza y Cuerpo (partes): se crean dentro de la clase Persona y no existen de manera independiente.
  - Si se elimina el objeto Persona, automáticamente también desaparecen Cabeza y Cuerpo.



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

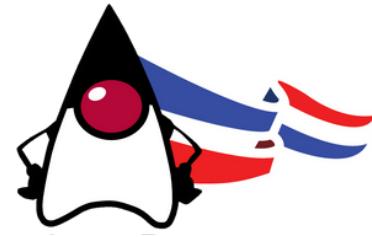
# Comparaciones / aclaraciones importantes

- **Agregación vs Composición:** muchas confusiones aquí. Lo clave es el ciclo de vida de las partes:
  - En composición: parte no tiene sentido sin todo; su existencia depende del todo.
  - En agregación: parte puede existir de manera independiente.
- **Asociación simple vs dependencia:**
  - Asociación indica que una clase tiene un atributo o relación persistente con otra.
  - Dependencia es más débil / temporal – por ejemplo “usa como parámetro” o “usa en un método local”.
- **Multiplicidad** se puede combinar con cualquiera de las relaciones que tienen extremos: asociación, agregación, composición. Siempre importante para dejar claro si “uno”, “muchos”, “cero o muchos”, etc.



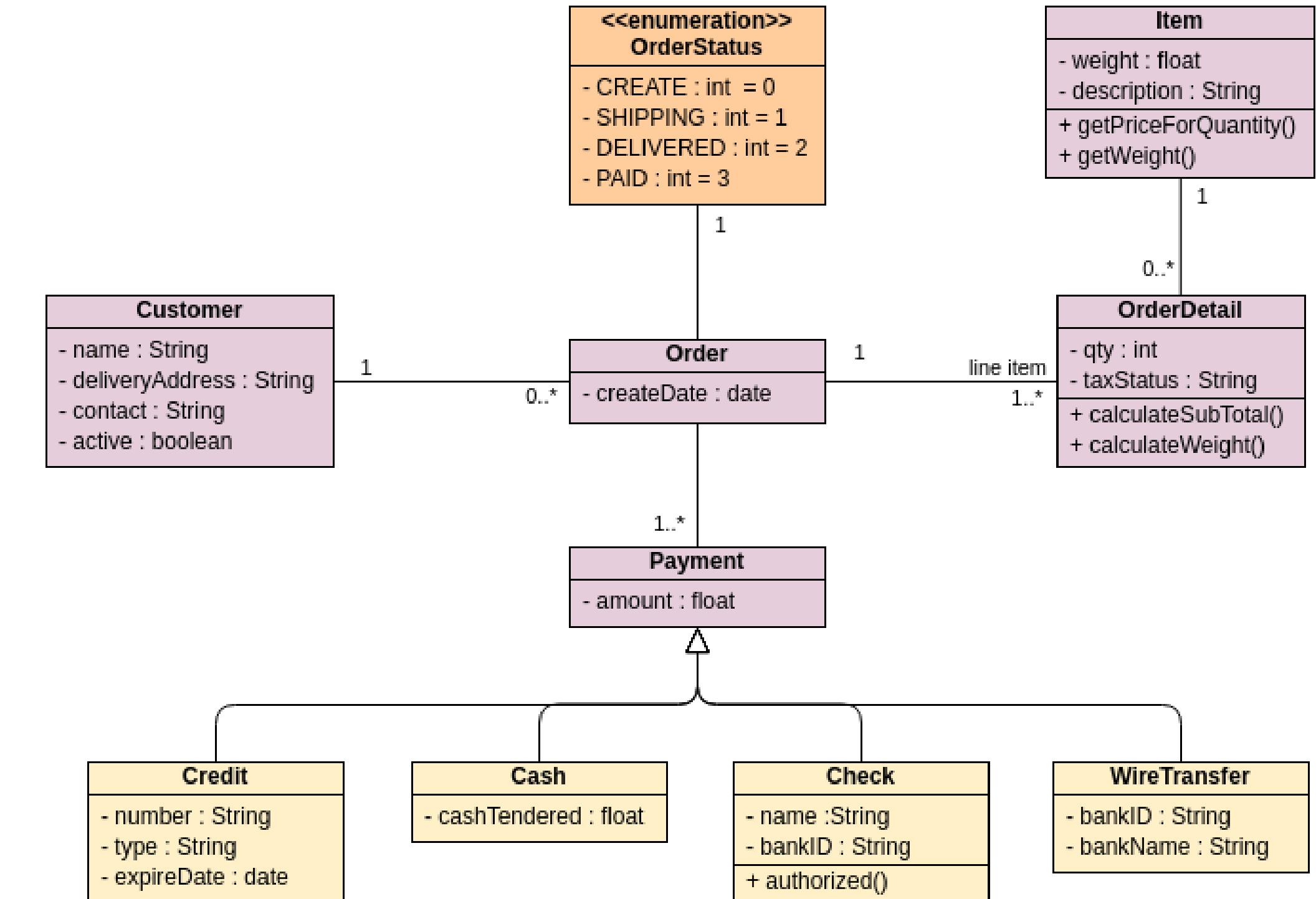
# Ventajas de usar diagramas de clases bien diseñadas

- Facilitan la comunicación entre equipo de desarrollo, analistas, arquitectos.
- Ayudan a detectar problemas de diseño temprano (acoplamientos indeseados, clases demasiado grandes, relaciones débiles, etc.).
- Mejoran la mantenibilidad: si entiendes claramente las relaciones, los cambios futuros serán menos costosos.
- Sirven de guía para la implementación (¿qué atributos?, ¿qué referencias debe tener cada clase?, ¿qué interfaces usar?).



# Diagrama de clases – Sistema de pedidos

El siguiente diagrama de clases modela un pedido de cliente de un catálogo minorista. La clase central es la Orden . Asociados a ella están el Cliente que realiza la compra y el Pago . Un pago es uno de cuatro tipos: efectivo , cheque , crédito o transferencia bancaria . El pedido contiene OrderDetails (artículos de línea), cada uno con su artículo asociado .



Fuente: <https://blog.visual-paradigm.com/es/what-are-the-six-types-of-relationships-in-uml-class-diagrams/>

→ Association

→ Inheritance

→ Implementation

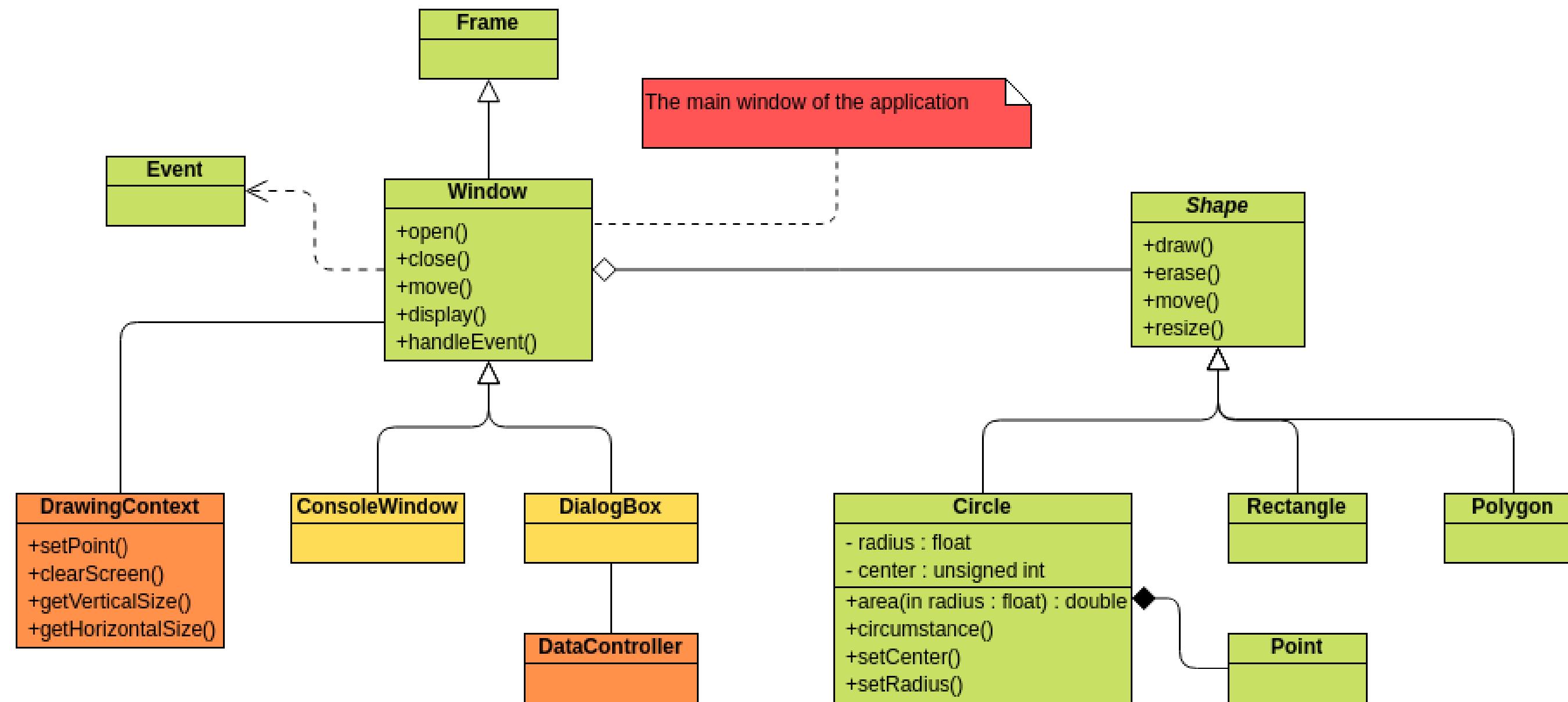
→ Dependency

→ Aggregation

→ Composition

# Ejemplo de diagrama de clases: GUI

Un diagrama de clases también puede tener notas adjuntas a clases o relaciones.



Fuente: <https://blog.visual-paradigm.com/es/what-are-the-six-types-of-relationships-in-uml-class-diagrams/>

→ **Association**

→ **Inheritance**

→ **Implementation**

→ **Dependency**

→ **Aggregation**

→ **Composition**

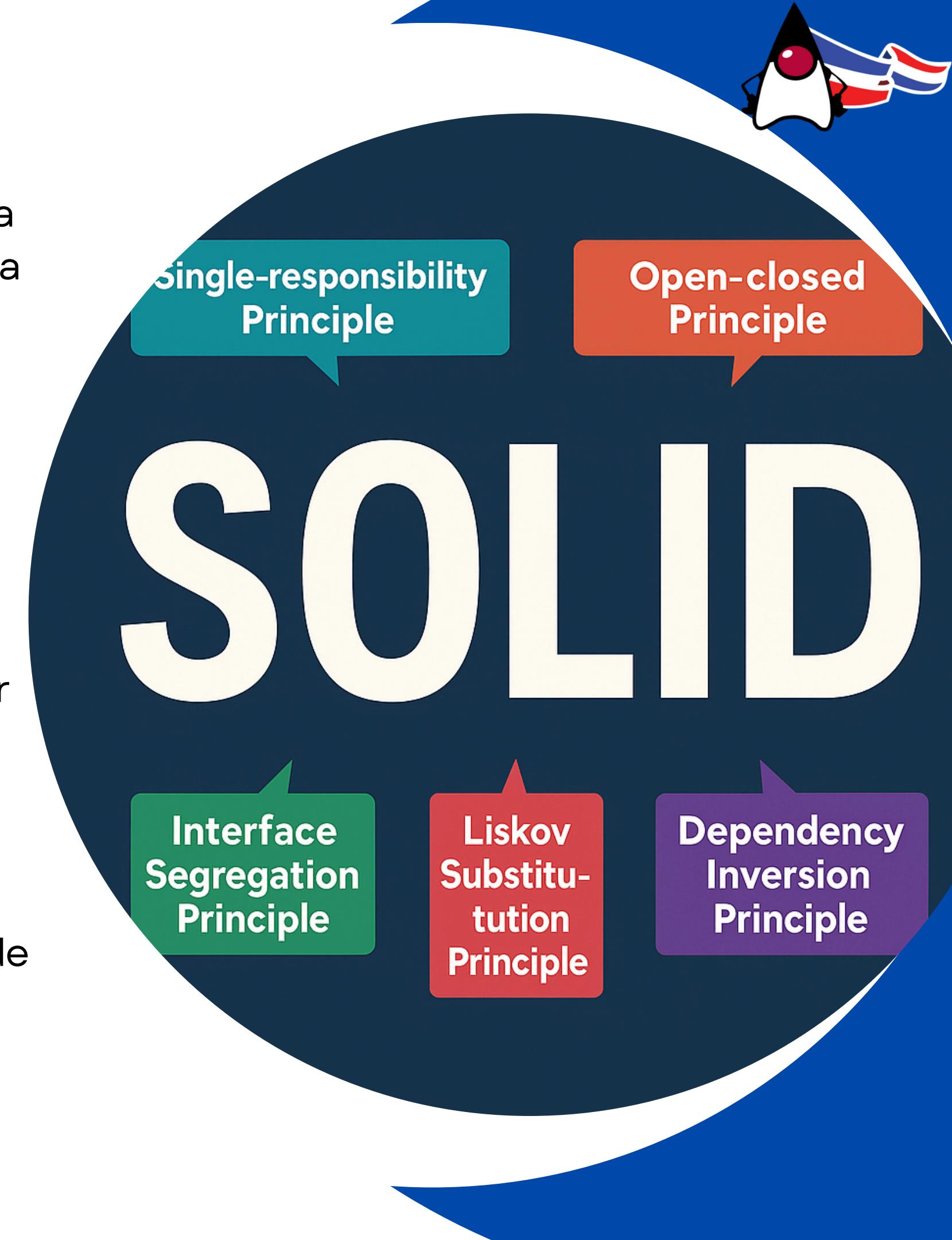


# Principios SOLID

**SOLID** es un conjunto de cinco principios de diseño orientado a objetos propuestos por **Robert C. Martin**. Su objetivo es guiar la creación de software más claro, mantenible, flexible y fácil de extender a medida que los proyectos crecen.

- **S (Single Responsibility)**: cada clase debe tener una sola responsabilidad o motivo de cambio.
- **O (Open/Closed)**: el código debe poder extenderse sin necesidad de modificar lo existente.
- **L (Liskov Substitution)**: las subclases deben poder sustituir a sus superclases sin romper el programa.
- **I (Interface Segregation)**: es mejor tener interfaces pequeñas y específicas, en lugar de una general con métodos innecesarios.
- **D (Dependency Inversion)**: los módulos deben depender de abstracciones, no de implementaciones concretas.

En conjunto, estos principios ayudan a crear software más mantenible, flexible y fácil de extender.





# Patrones de Diseño

## ¿Qué es un patrón de diseño?

- Los patrones de diseño (design patterns) son soluciones habituales a problemas comunes en el diseño de software. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular de tu código.
- A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos
- Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.



# Patrones de Diseño

## ¿En qué consiste el patrón?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

- El **propósito** del patrón explica brevemente el problema y la solución.
- La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
- La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.



# Patrones de Diseño

## ¿Por qué debería aprender sobre patrones?

La realidad es que podrías trabajar durante años como programador sin conocer un solo patrón. Mucha gente lo hace. Incluso en ese caso, podrías estar implementando patrones sin saberlo. Así que, ¿por qué dedicar tiempo a aprenderlos?

- Son un juego de herramientas de soluciones comprobadas a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.
- Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicaros de forma más eficiente. Podrías decir: “Oh, utiliza un singleton para eso”, y todos entenderían la idea de tu sugerencia. No habría necesidad de explicar qué es un singleton si conocen el patrón y su nombre.



# Patrones de Diseño

## Clasificación de los patrones

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña.

Los patrones más básicos y de más bajo nivel suelen llamarse idioms. Normalmente se aplican a un único lenguaje de programación.

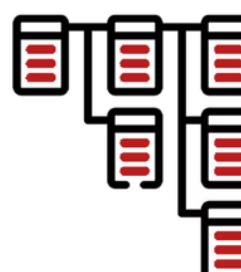
Los patrones más universales y de más alto nivel son los patrones de arquitectura.

Además, todos los patrones pueden clasificarse por su propósito.

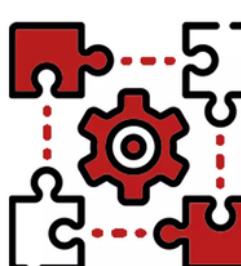


### Creacionales:

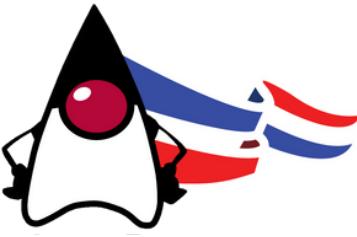
Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.



**Estructurales:** Explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.



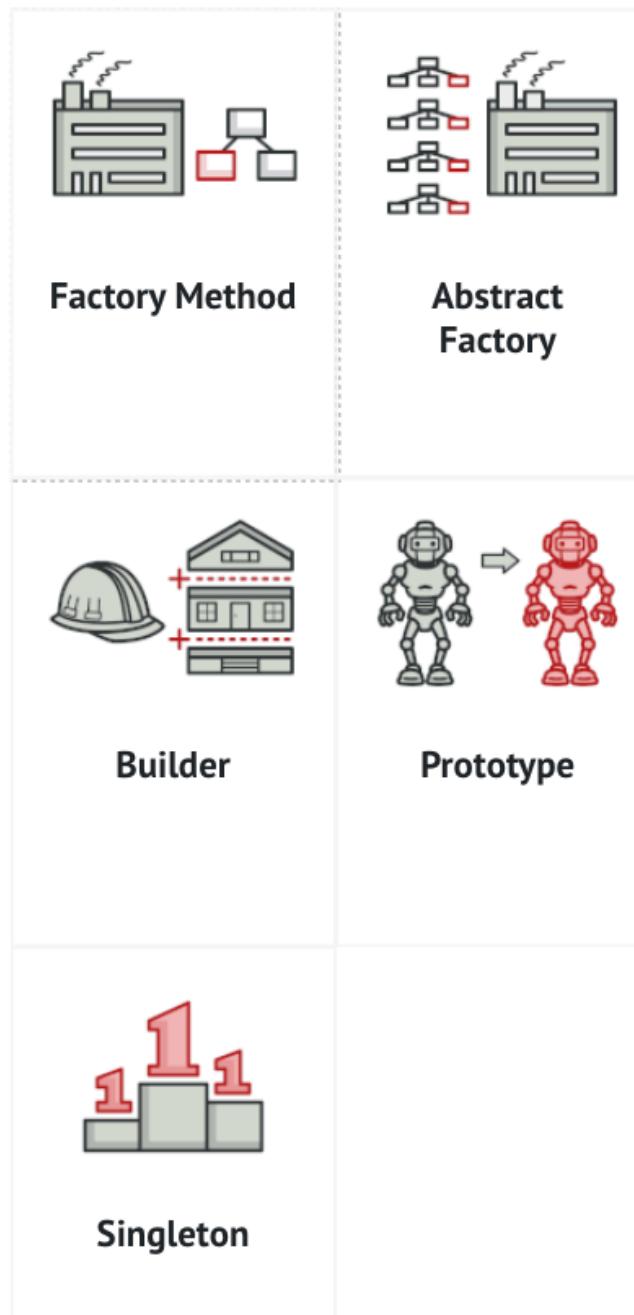
**Comportamiento:** Se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.



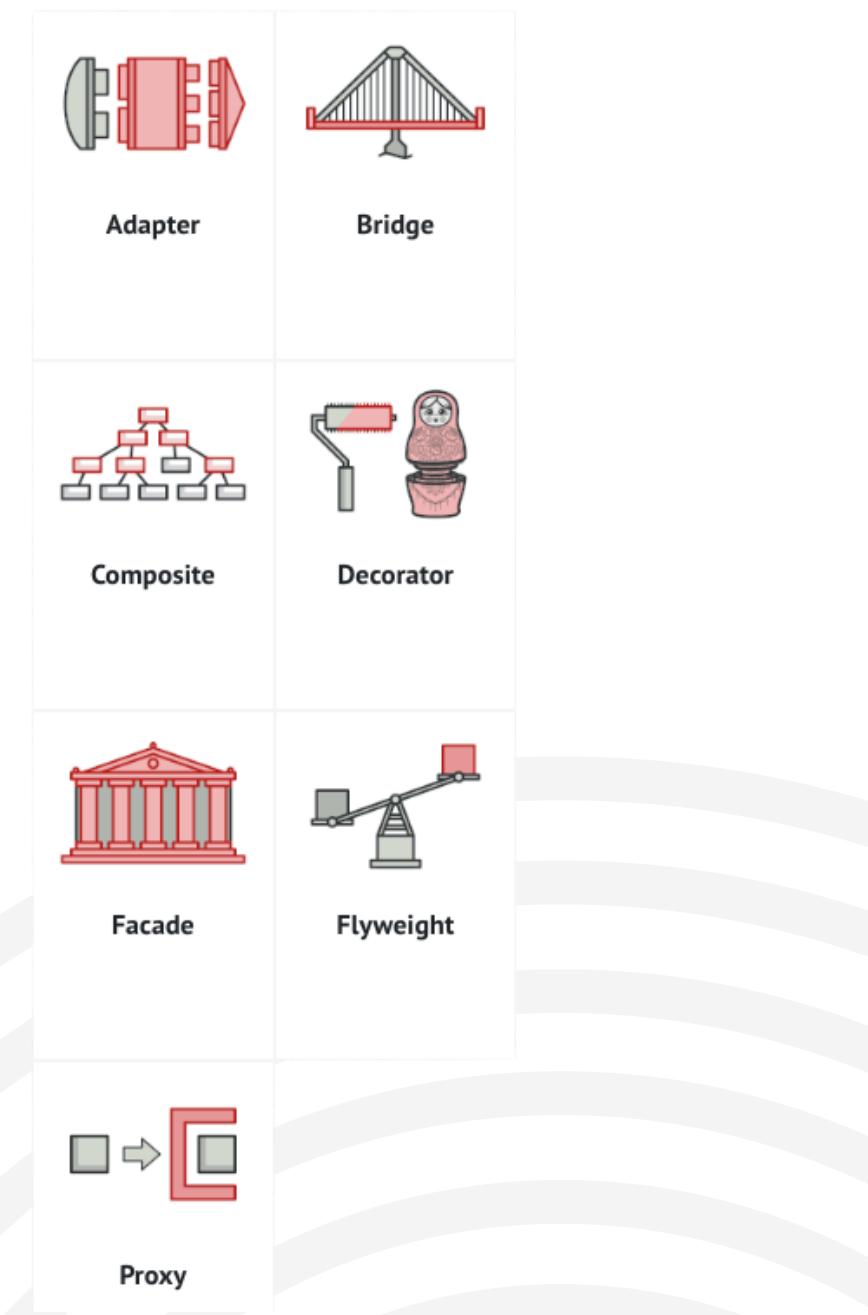
# Patrones de Diseño

## Clasificación de los patrones

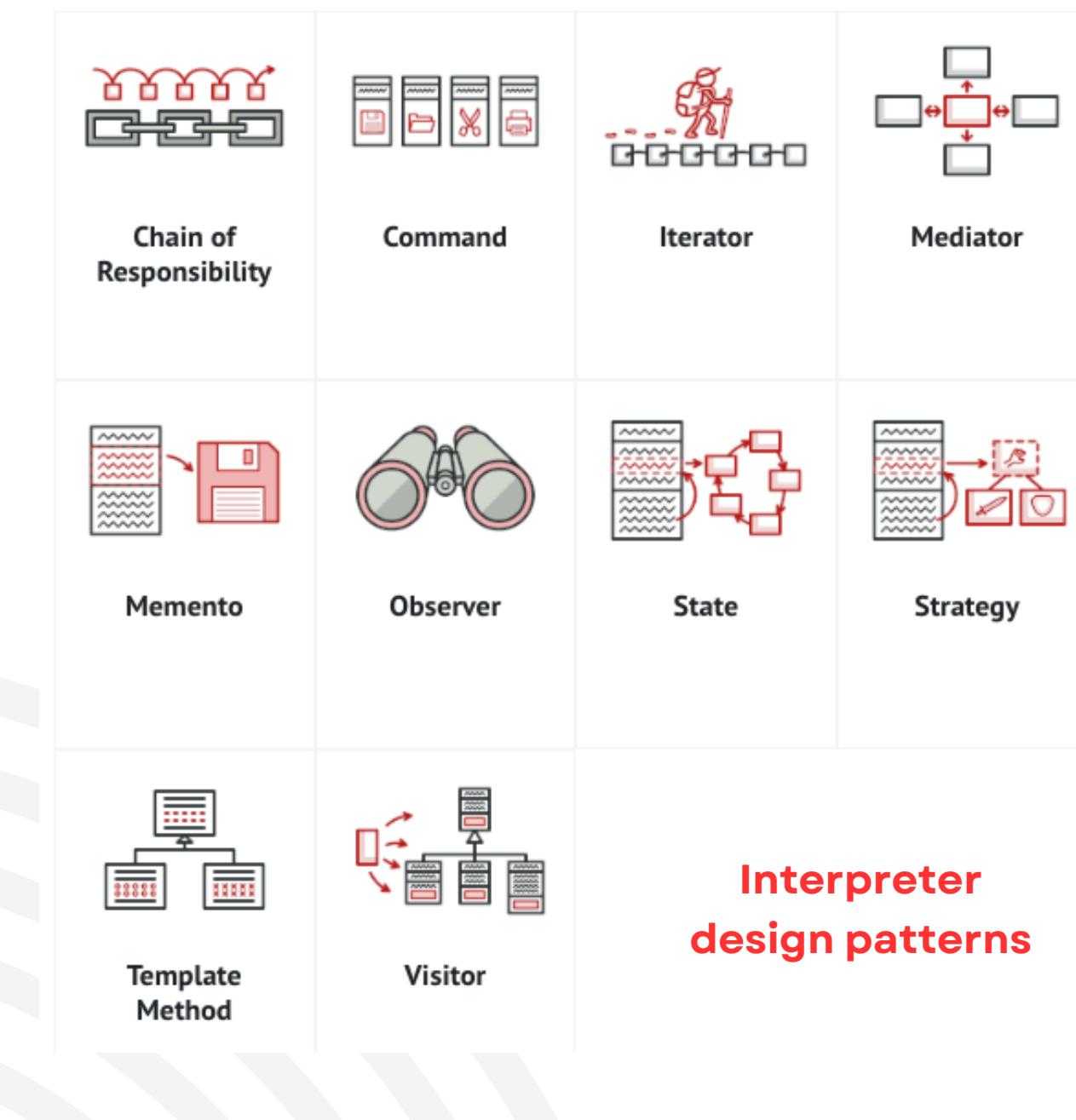
### Patrones creacionales



### Patrones estructurales



### Patrones de comportamiento

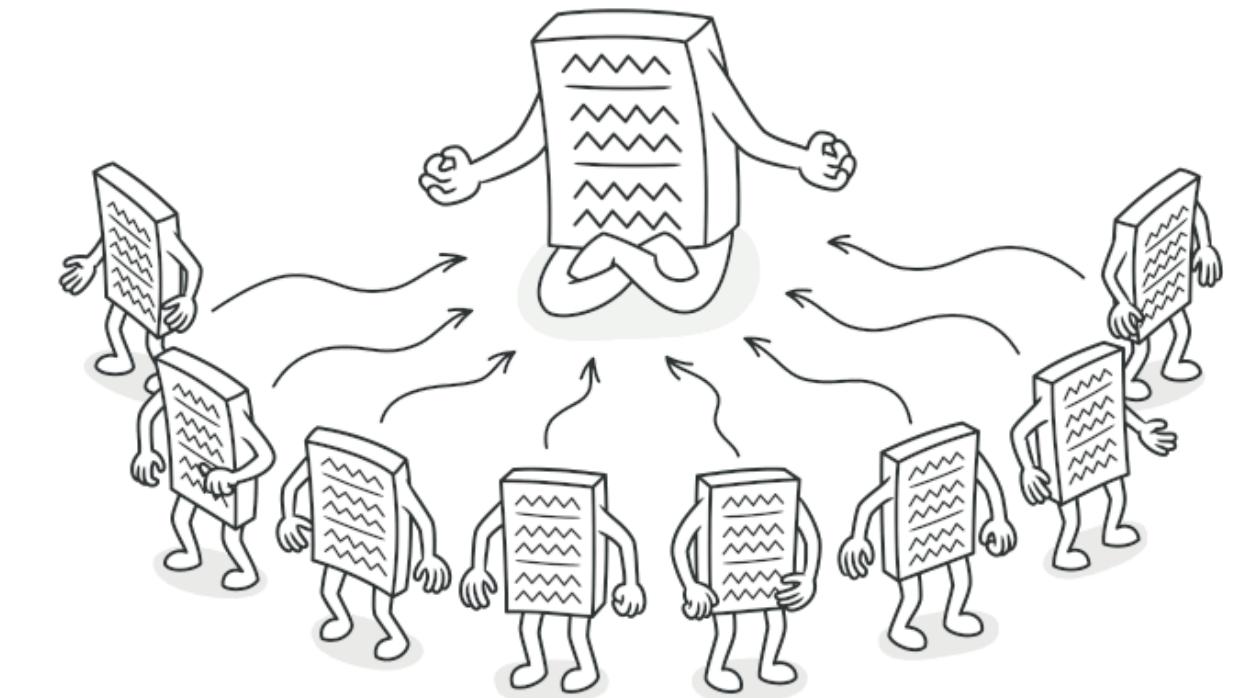


# Patrones de Diseño - Singleton

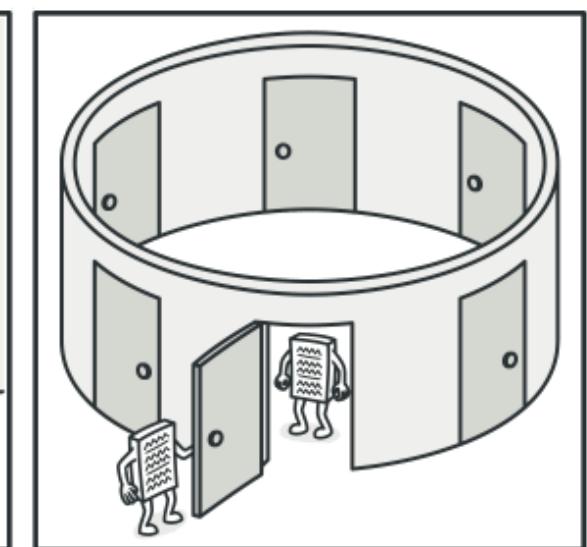
Patrones creacionales - Instancia única



**Propósito:** Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



**Problema:** El patrón Singleton asegura una sola instancia de una clase para controlar recursos compartidos, aunque viola el principio de responsabilidad única.



# Patrones de Diseño - Singleton

Patrones creacionales - Instancia única



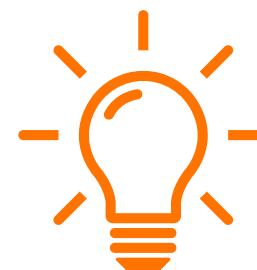
**Solución:** El patrón Singleton oculta su constructor y usa un método estático que crea una sola instancia y siempre devuelve el mismo objeto en llamadas posteriores.



**Analogía en el mundo real:** El gobierno es un ejemplo excelente del patrón Singleton. Un país sólo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título “Gobierno de X” es un punto de acceso global que identifica al grupo de personas a cargo.

# Patrones de Diseño - Singleton

Patrones creacionales - Instancia única



## Aplicabilidad:

- Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.
- Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

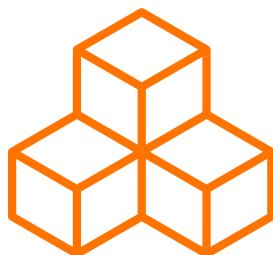


## Pros y contras:

- Asegura una sola instancia, acceso global y carga diferida.
- X Viola SRP, puede ocultar mal diseño, es complejo en hilos, y difícil de probar con mocks.

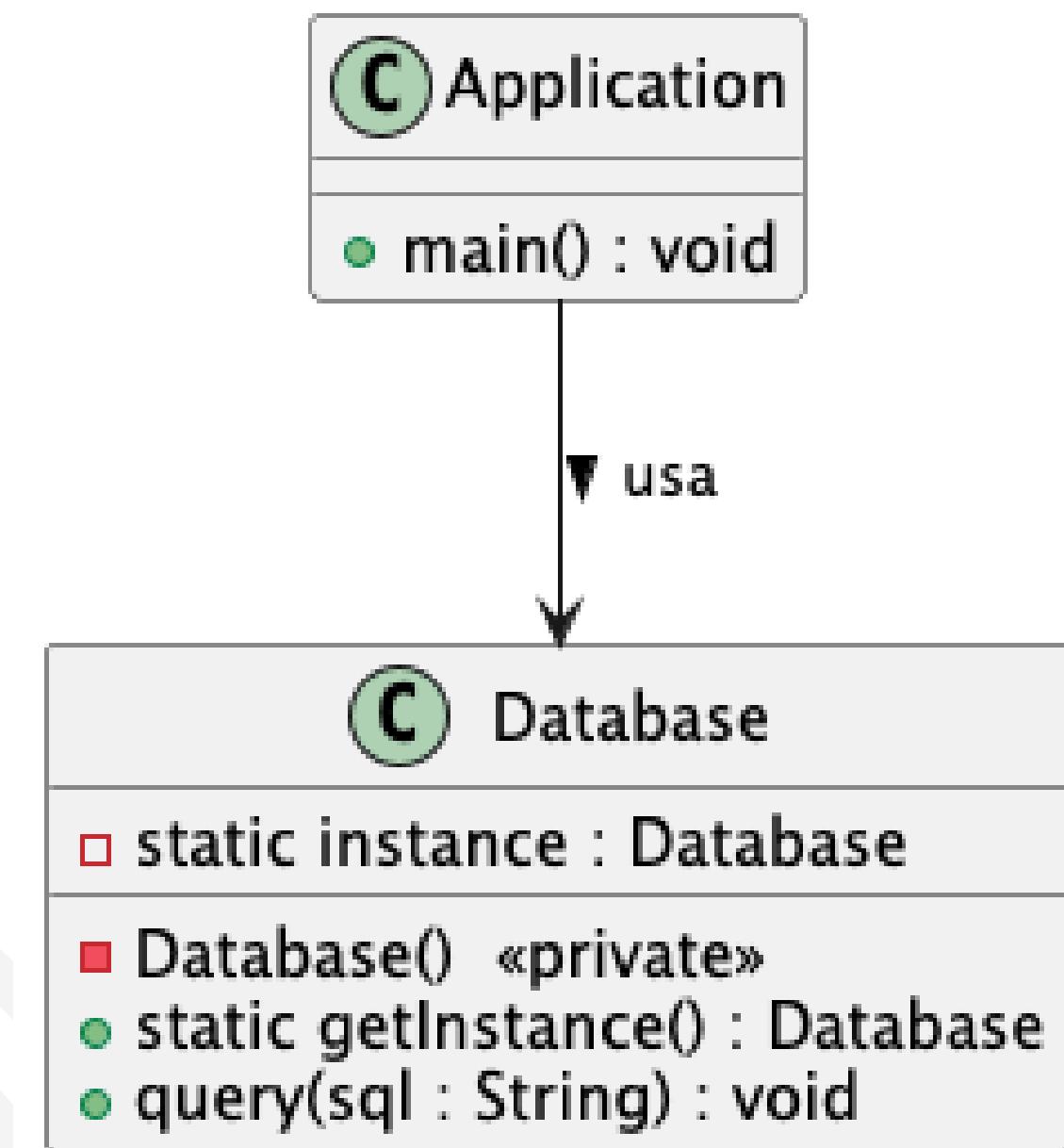
# Patrones de Diseño - Singleton

Patrones creacionales - Instancia única



**Estructura:** En este ejemplo, la clase de conexión de la base de datos actúa como Singleton. Esta clase no tiene un constructor público, por lo que la única manera de obtener su objeto es invocando el método `obtenerInstancia`. Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes.

- La clase Singleton declara el método estático `obtenerInstancia` que devuelve la misma instancia de su propia clase.
- El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

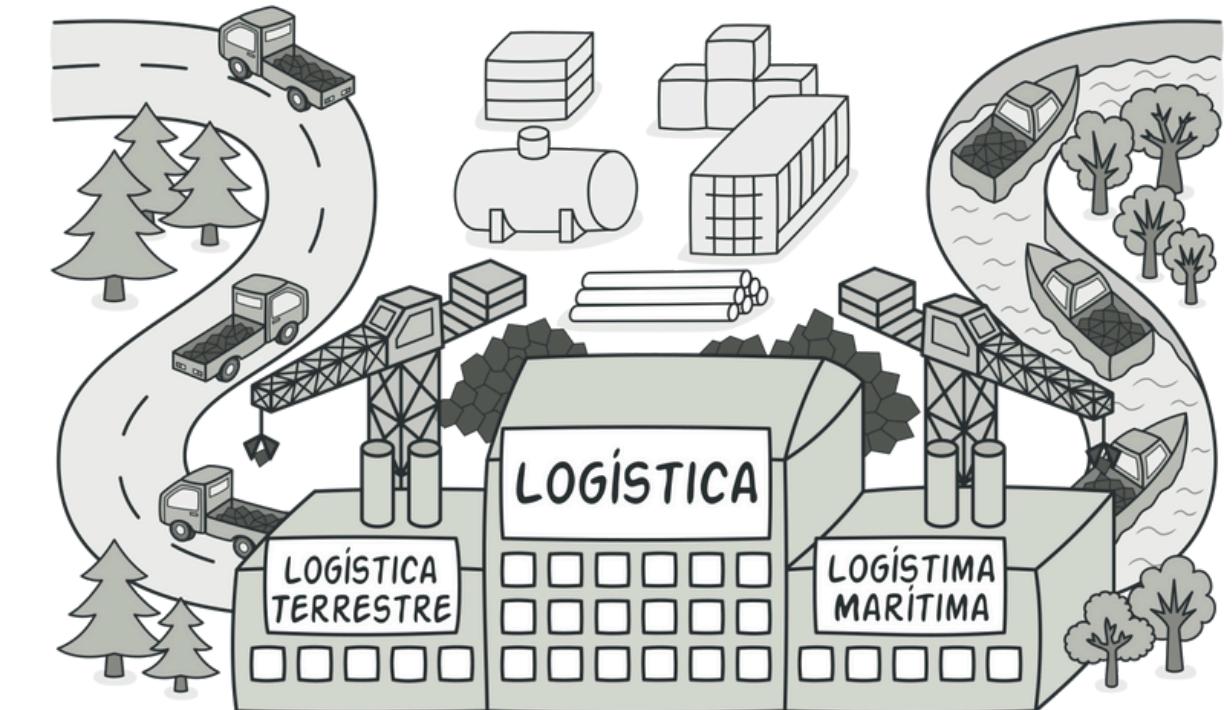
→ Composition

# Patrones de Diseño - Factory Method

Patrones creacionales - Método fábrica, Constructor virtual

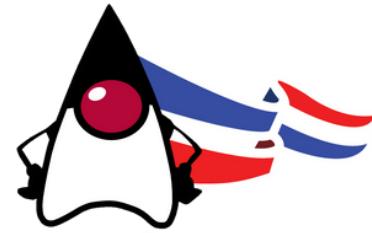


**Propósito:** Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



**Problema:** Código de logística acoplado a **Camión**: añadir **Barco** u otros medios exige tocar toda la base, proliferan condicionales y crece el acoplamiento, bajando la mantenibilidad.



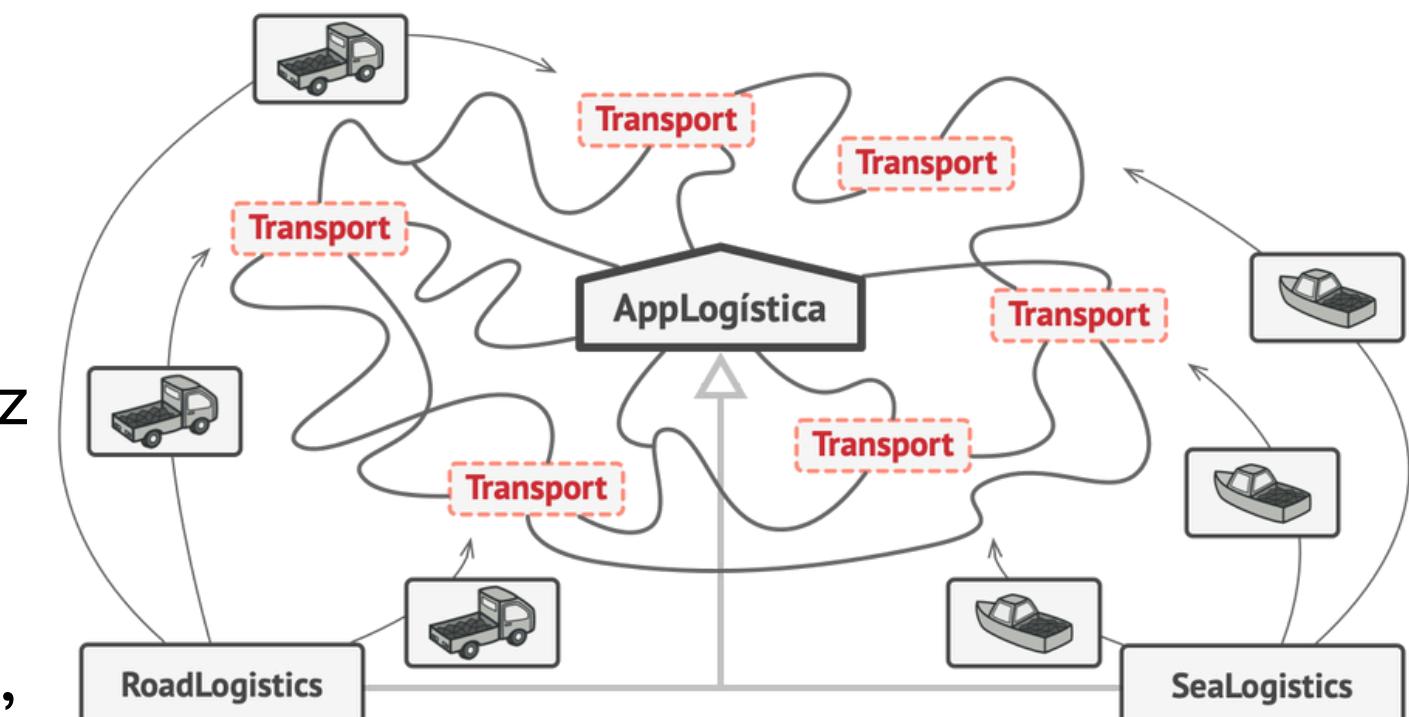


# Patrones de Diseño - Factory Method

Patrones creacionales - Método fábrica, Constructor virtual



**Solución:** El patrón Factory Method propone crear objetos mediante un método fábrica en lugar de instancias directas con **new**. Esto permite que subclases decidan qué producto instanciar, siempre que comparten una interfaz común, como Transporte con el método **entrega**. Así, LogísticaTerrestre devuelve camiones y LogísticaMarítima devuelve barcos, sin que el cliente sepa los detalles. El código cliente sólo depende de la interfaz, no de implementaciones concretas, logrando flexibilidad, menor acoplamiento y facilidad para ampliar con nuevos tipos de transporte.



# Patrones de Diseño - Factory Method

Patrones creacionales - Método fábrica, Constructor virtual



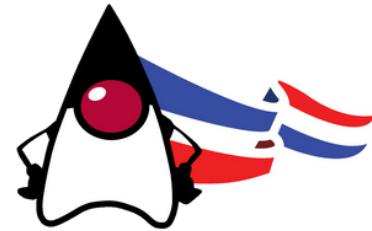
## Aplicabilidad:

- Utiliza el Método Fábrica cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.
- Utiliza el Factory Method cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.
- Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.



## Pros y contras:

- Menos acoplamiento, SRP y abierto/cerrado..
- Más complejidad: requiere crear varias subclases para nuevos productos.

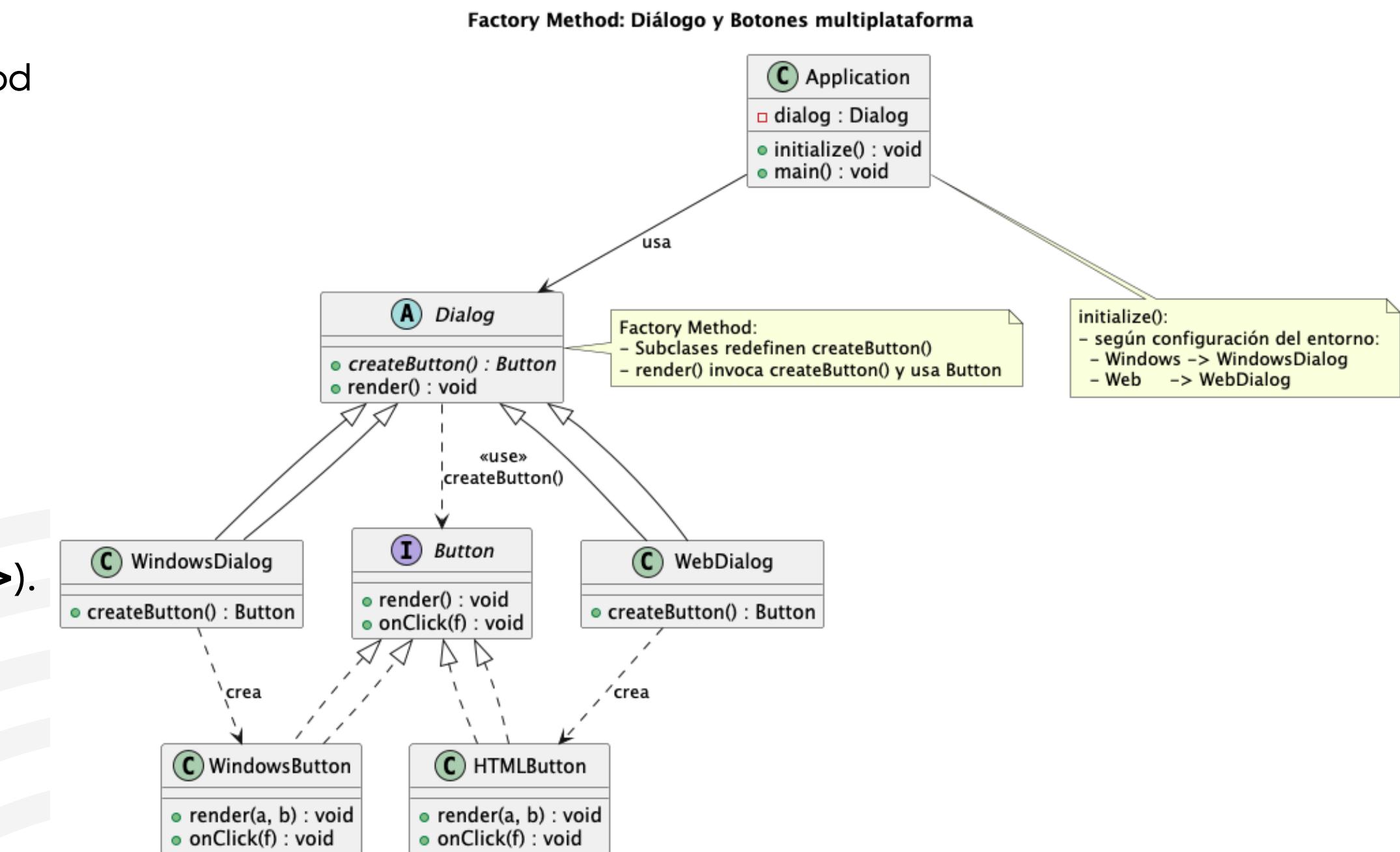


# Patrones de Diseño - Factory Method

Patrones creacionales - Método fábrica, Constructor virtual



- **Estructura:**
- **Dialog** (abstracta) declara el Factory Method **createButton()** y la lógica **render()** que trabaja contra la interfaz **Button**.
- **WindowsDialog** y **WebDialog** heredan de **Dialog** y crean (..>) sus botones concretos: **WindowsButton** y **HTMLButton**, respectivamente.
- **Button** es la interfaz de producto; sus implementaciones concretas la realizan (..|>).
- **Application** es el cliente que decide qué creador usar (según config) y luego llama **dialog.render()** sin acoplarse a clases concretas de botones.



→ Association

→ Dependency

→ Inheritance

→ Aggregation

→ Implementation

→ Composition

# Patrones de Diseño - Abstract Factory

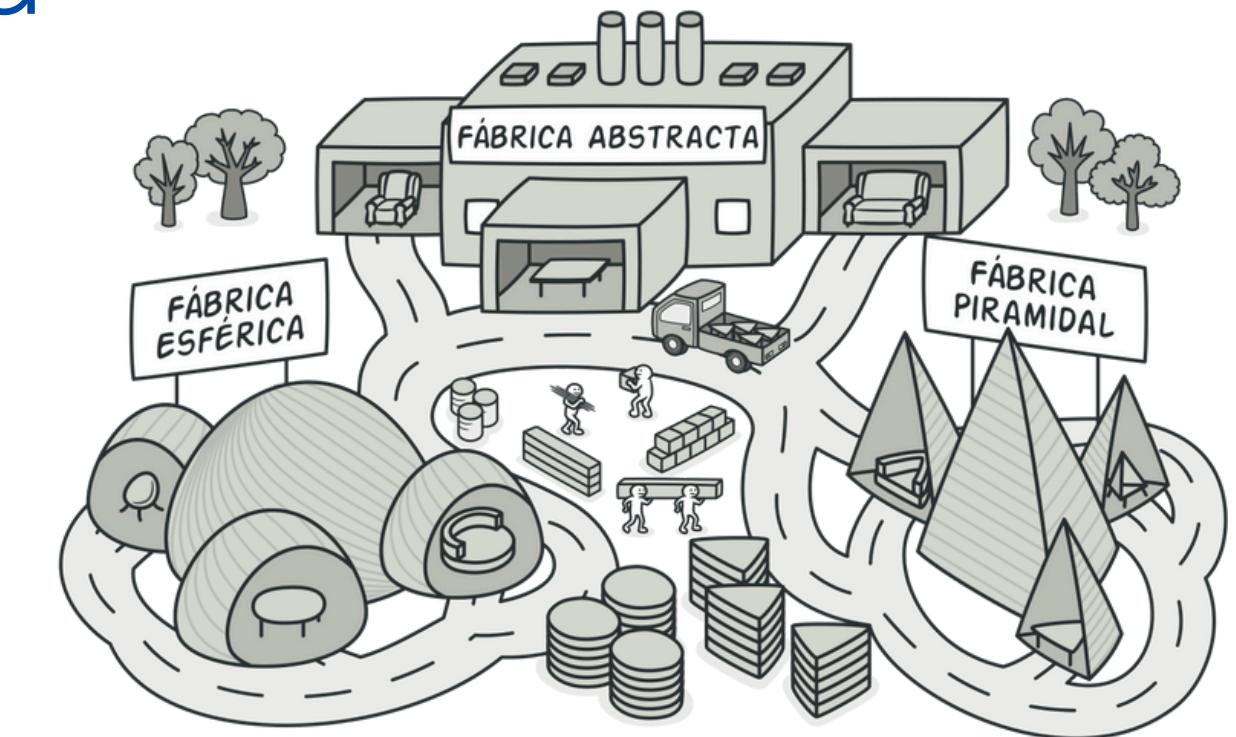
## Patrones creacionales - Fábrica abstracta

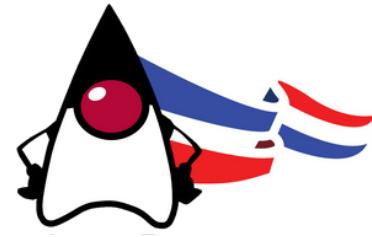


**Propósito:** Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



**Problema:** Crear familias de muebles (Silla, Sofá, Mesilla) con variantes (Moderna, Victoriana, ArtDecó) que combinen entre sí. Y poder añadir nuevas familias/variantes sin modificar el código cliente ni romper el sistema.



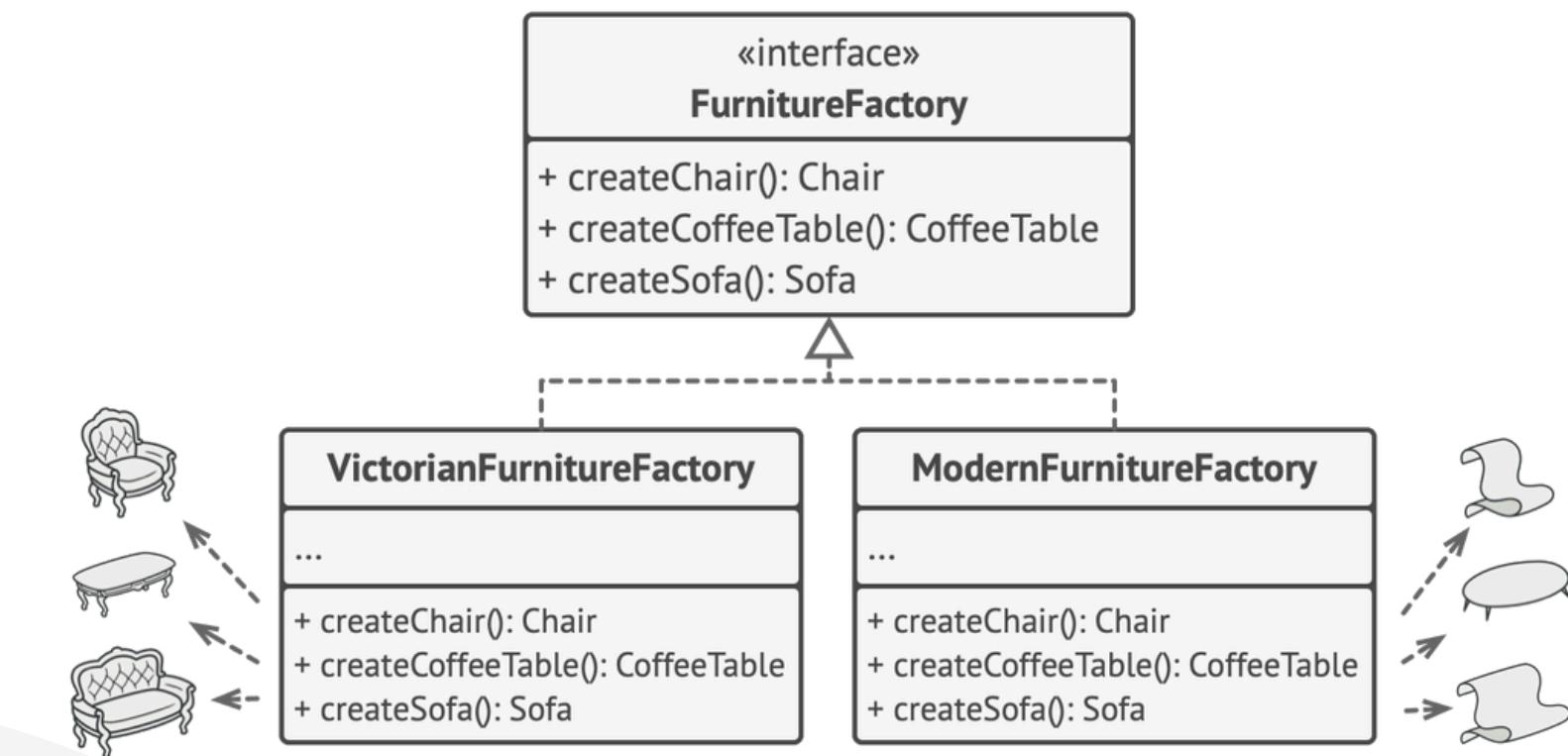


# Patrones de Diseño - Abstract Factory

## Patrones creacionales - Fábrica abstracta



**Solución:** El patrón Abstract Factory define interfaces para cada producto de una familia (Silla, Sofá, Mesilla) y una fábrica abstracta con métodos de creación. Cada variante (Moderna, Victoriana, ArtDecó) implementa su propia fábrica concreta que devuelve productos combinados coherentemente. El cliente interactúa solo con interfaces abstractas, sin conocer las clases concretas ni su lógica interna. Así, puede cambiar la fábrica y obtener distintas variantes sin alterar su código. Normalmente, la fábrica concreta se crea en la inicialización según configuración o entorno. Este enfoque asegura consistencia entre productos, bajo acoplamiento y gran extensibilidad..



# Patrones de Diseño - Abstract Factory

## Patrones creacionales - Fábrica abstracta



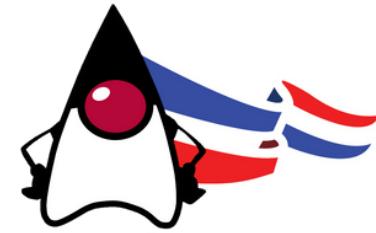
### Aplicabilidad:

- Utiliza el patrón Abstract Factory cuando tu código deba funcionar con varias familias de productos relacionados, pero no deseas que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.
- Considera la implementación del patrón Abstract Factory cuando tengas una clase con un grupo de métodos de fábrica que nublen su responsabilidad principal.



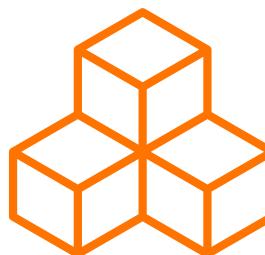
### Pros y contras:

-  Productos siempre compatibles, menos acoplamiento, SRP y abierto/cerrado.
-  Aumenta la complejidad: requiere muchas interfaces y clases adicionales.



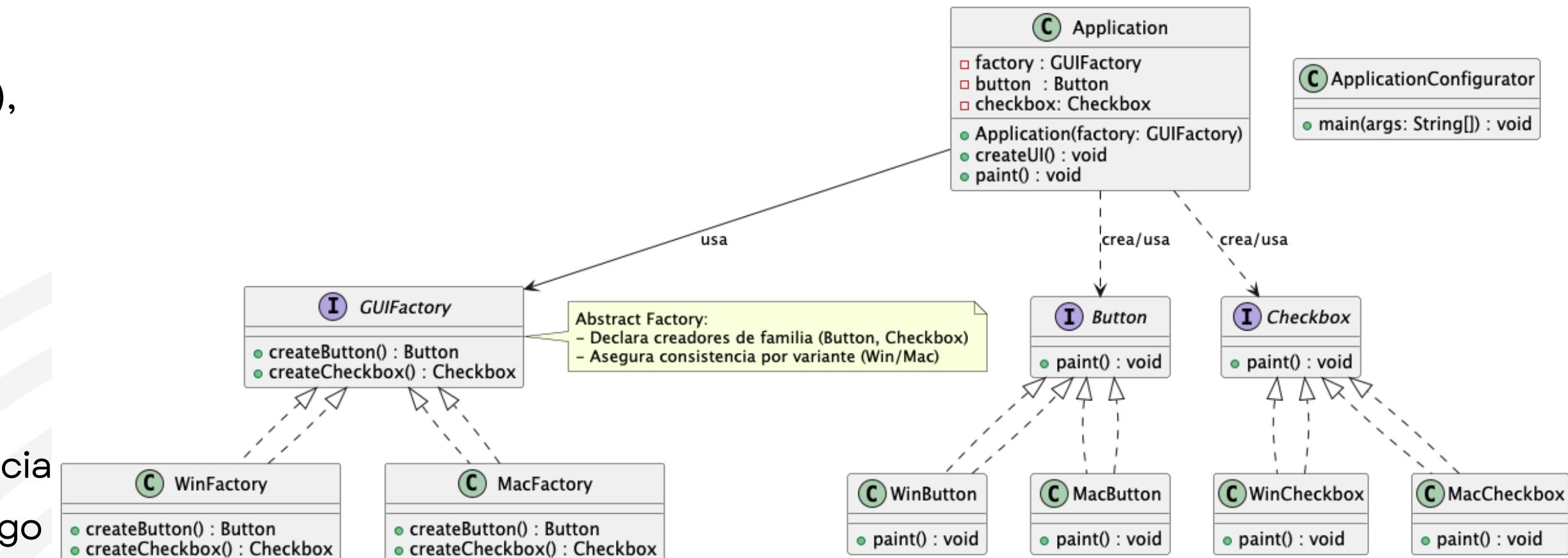
# Patrones de Diseño - Abstract Factory

## Patrones creacionales - Fábrica abstracta



- **Estructura:**
- **GUIFactory** define el contrato para crear una familia de productos relacionados: **Button** y **Checkbox**.
- **WinFactory** y **MacFactory** son fábricas concretas que crean productos concretos (**WinButton/WinCheckbox** y **MacButton/MacCheckbox**) compatibles entre sí (misma variante SO).
- **Application** es el cliente que depende solo de abstracciones (**GUIFactory**, **Button**, **Checkbox**), por lo que no se acopla a implementaciones concretas.
- Cambiar la variante (Windows/Mac) se hace en la inicialización (no en la lógica de negocio), manteniendo coherencia visual/funcional sin tocar el código cliente.

Abstract Factory: GUI multiplataforma (Windows / Mac)



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

# Patrones de Diseño - Builder

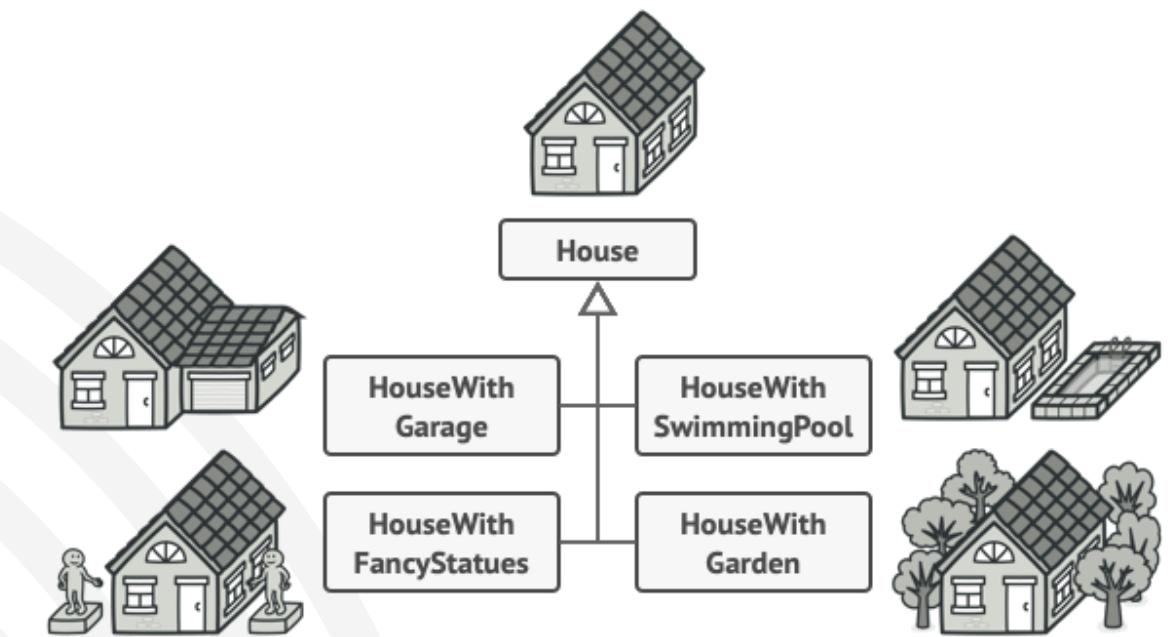
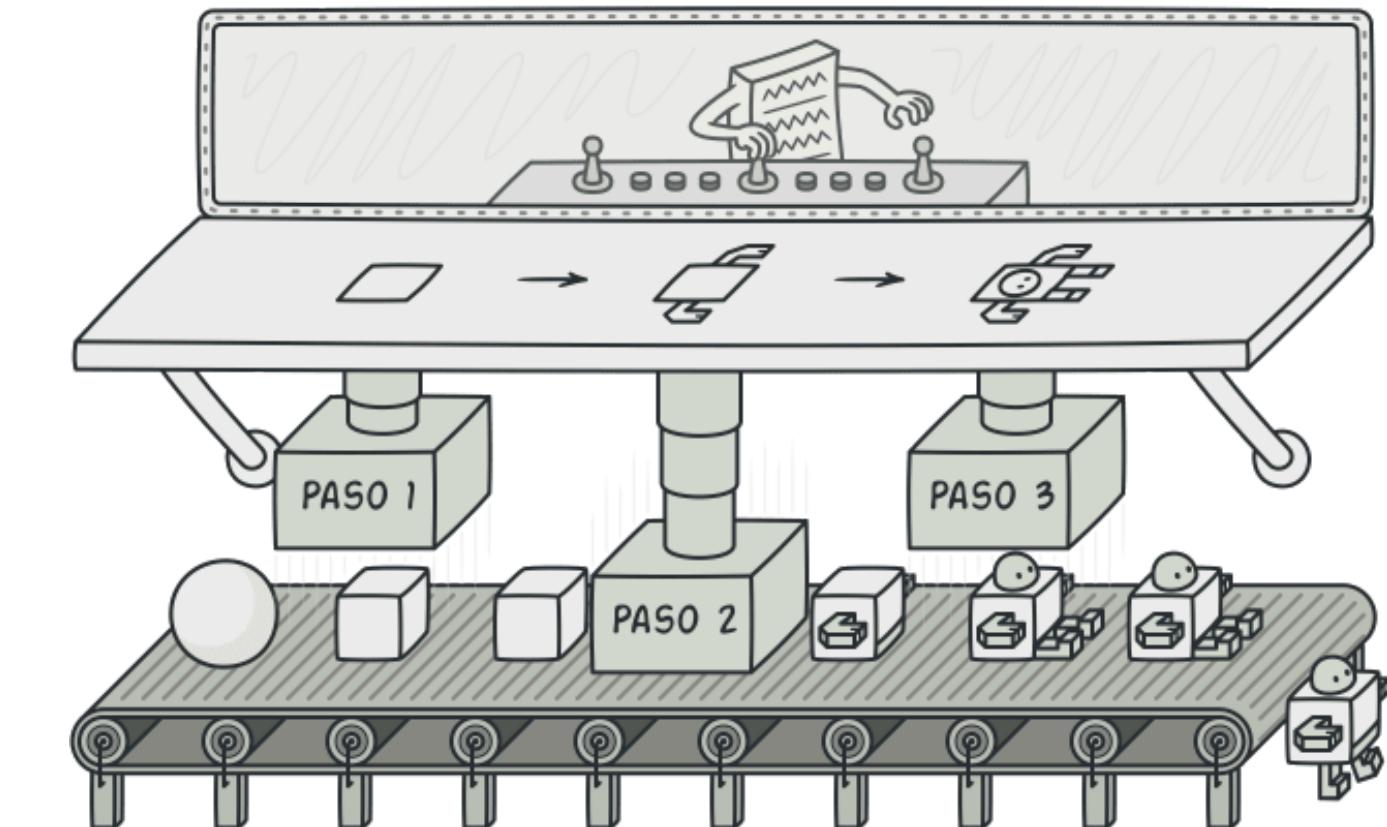
## Patrones creacionales - Constructor



**Propósito:** 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:** Objetos complejos pueden requerir muchos pasos o parámetros. Crear subclases por cada variante complica la jerarquía; usar un constructor enorme genera llamadas confusas y parámetros inútiles.

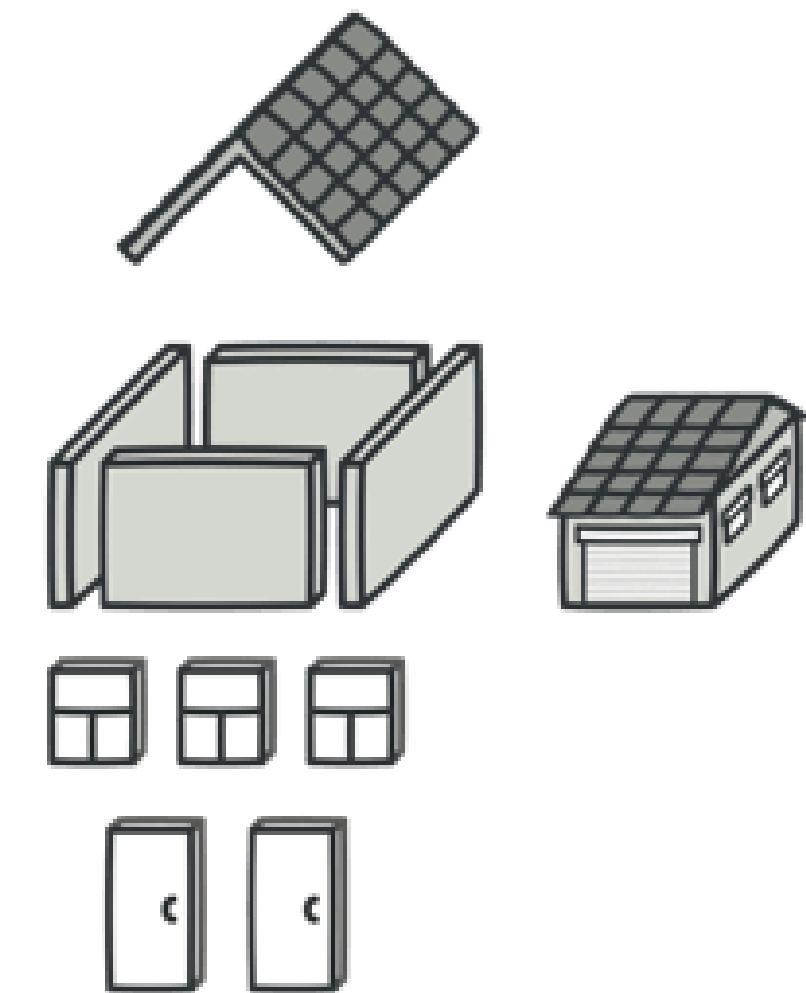
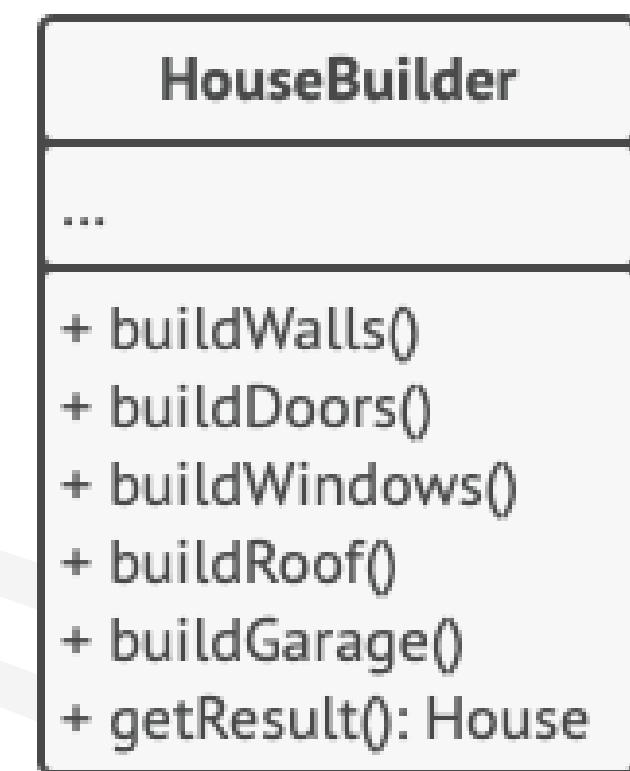


# Patrones de Diseño - Builder

## Patrones creacionales - Constructor



**Solución:** El patrón Builder separa la construcción de un objeto complejo en objetos llamados constructores. Permite crear productos paso a paso, ejecutando solo los pasos necesarios (ej. paredes, puertas, tejado). Cada constructor puede implementar los pasos de forma distinta: madera y vidrio para una cabaña, piedra y hierro para un castillo o materiales lujosos para un palacio. Así, con la misma secuencia de pasos se obtienen diferentes representaciones, siempre que los constructores comparten una interfaz común. Opcionalmente, una clase directora organiza el orden de construcción, reutilizando rutinas y ocultando los detalles al cliente, que solo recibe el producto terminado.



# Patrones de Diseño - Builder

## Patrones creacionales - Constructor



### Aplicabilidad:

- Utiliza el patrón Builder para evitar un “constructor telescopico”.
- 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).
- Utiliza el patrón Builder para construir árboles con el patrón Composite u otros objetos complejos.



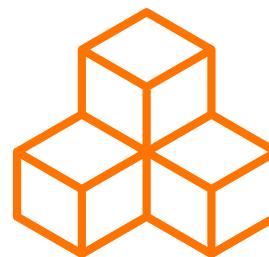
### Pros y contras:

- Builder permite crear objetos paso a paso, reutilizar código y aislar la construcción compleja.
- Aumenta la complejidad: exige definir varias clases adicionales.

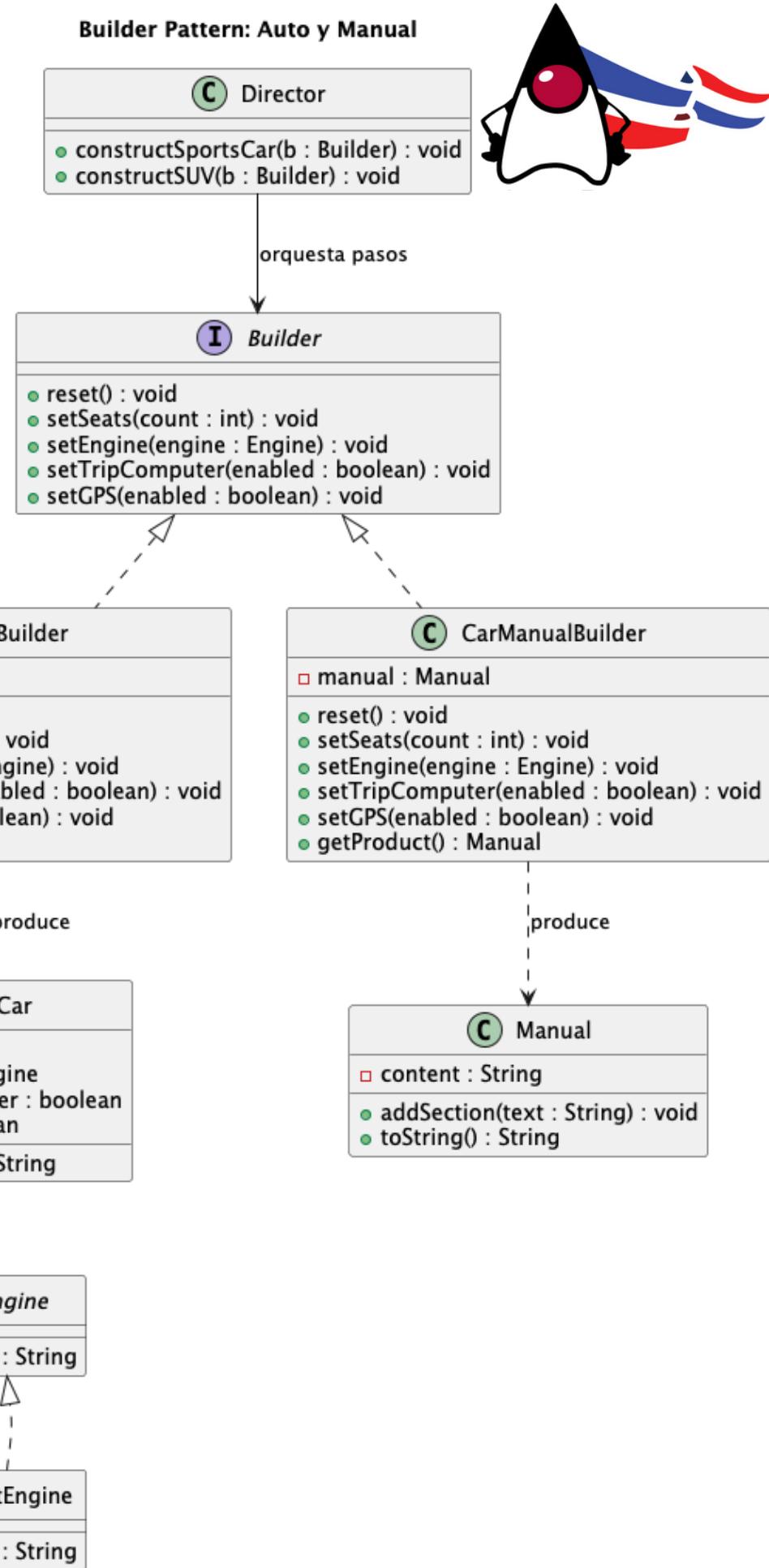


# Patrones de Diseño - Builder

## Patrones creacionales - Constructor



- **Estructura:**
- **Builder** define los pasos de construcción para una familia de productos.
- **CarBuilder** y **CarManualBuilder** implementan esos pasos, pero producen objetos diferentes (**Car** y **Manual**).
- **Director** orquesta el orden de los pasos para construir configuraciones conocidas (ej. deportivo, SUV).
- El resultado se obtiene del builder concreto (**getProduct()**), evitando acoplar el director a un producto específico.



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

# Patrones de Diseño - Prototype

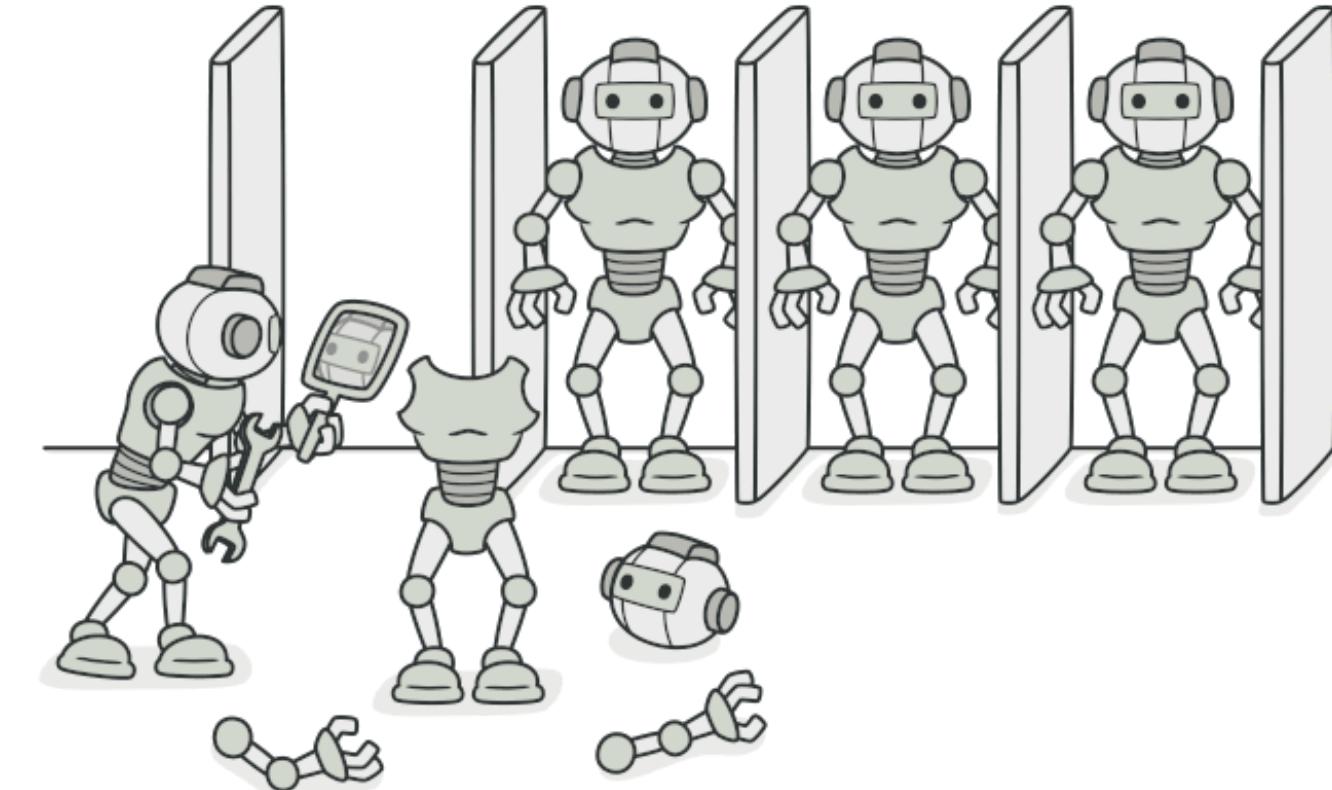
## Patrones creacionales - Prototipo, Clone



**Propósito:** Es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.



**Problema:** Para clonar un objeto hay que crear otro de la misma clase y copiar sus campos. El problema es que algunos son privados, y además el código queda acoplado a la clase concreta o puede que solo conozca su interfaz.



# Patrones de Diseño - Prototype

## Patrones creacionales - Prototipo, Clone



**Solución:** El patrón Prototype delega la clonación al propio objeto, definiendo una interfaz común con un método **clonar**. Así se pueden copiar objetos sin acoplar el código a clases concretas, incluso copiando campos privados. Los objetos que implementan esta interfaz se llaman prototipos. Su método crea una nueva instancia con los mismos valores del original. Este patrón es útil cuando hay muchos campos y configuraciones posibles, evitando crear subclases para cada caso. Además, permite trabajar con prototipos prefabricados: objetos ya configurados que pueden clonarse cuando se necesite, en lugar de construirlos desde cero, reduciendo complejidad y esfuerzo.



# Patrones de Diseño - Prototype

## Patrones creacionales - Prototipo, Clone



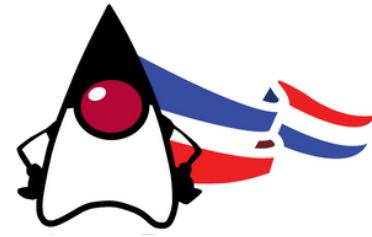
### Aplicabilidad:

- Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.
- Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.



### Pros y contras:

- Prototype permite clonar sin acoplar a clases, evita inicialización repetida y facilita crear objetos complejos.
- Puede complicarse al clonar objetos con referencias circulares.

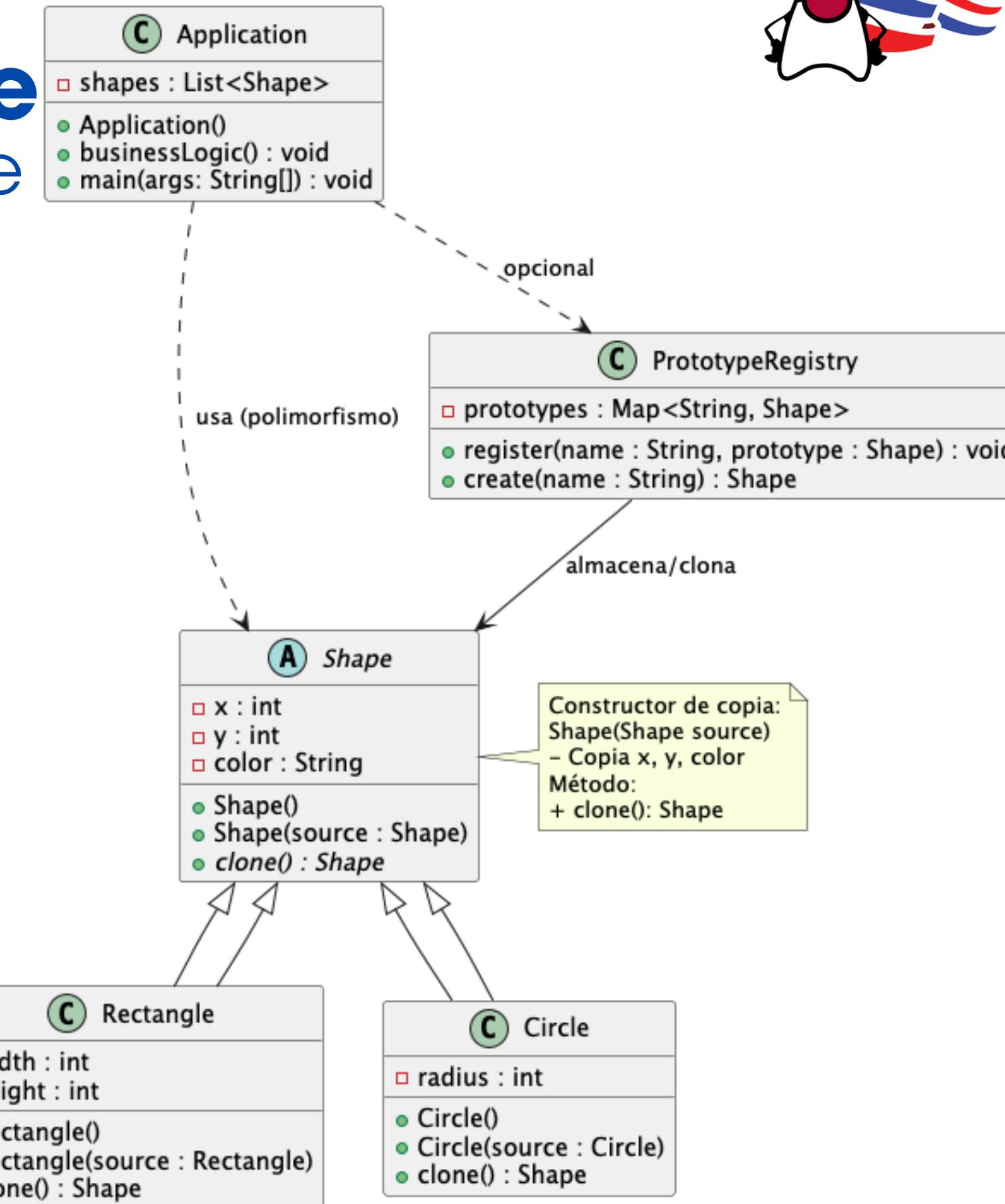


# Patrones de Diseño - Prototype

## Patrones creacionales - Prototipo, Clone



- **Estructura:**
- **Shape** es el prototipo base con un constructor de copia y un método abstracto **clone()**.
- **Rectangle** y **Circle** son prototipos concretos que implementan delegando en constructores de copia.
- **PrototypeRegistry** mantiene un catálogo (Map) **name → proto** entrega clones a pedido (**create(name)**).
- **Application** demuestra:
  - Clonado polimórfico (sin conocer la clase concreta).
  - Uso opcional del registro para obtener clones.



→ Association

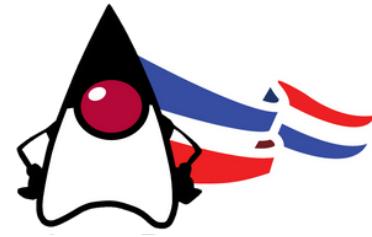
→ Inheritance

→ Implementation

→ Dependency

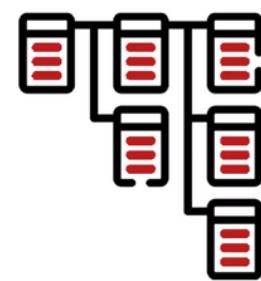
→ Aggregation

→ Composition



# Patrones de Diseño

## Clasificación de los patrones



**Estructurales:** Explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

<b>Adapter</b>	<b>Bridge</b>
<b>Composite</b>	<b>Decorator</b>
<b>Facade</b>	<b>Flyweight</b>
	<b>Proxy</b>

# Patrones de Diseño - Adapter

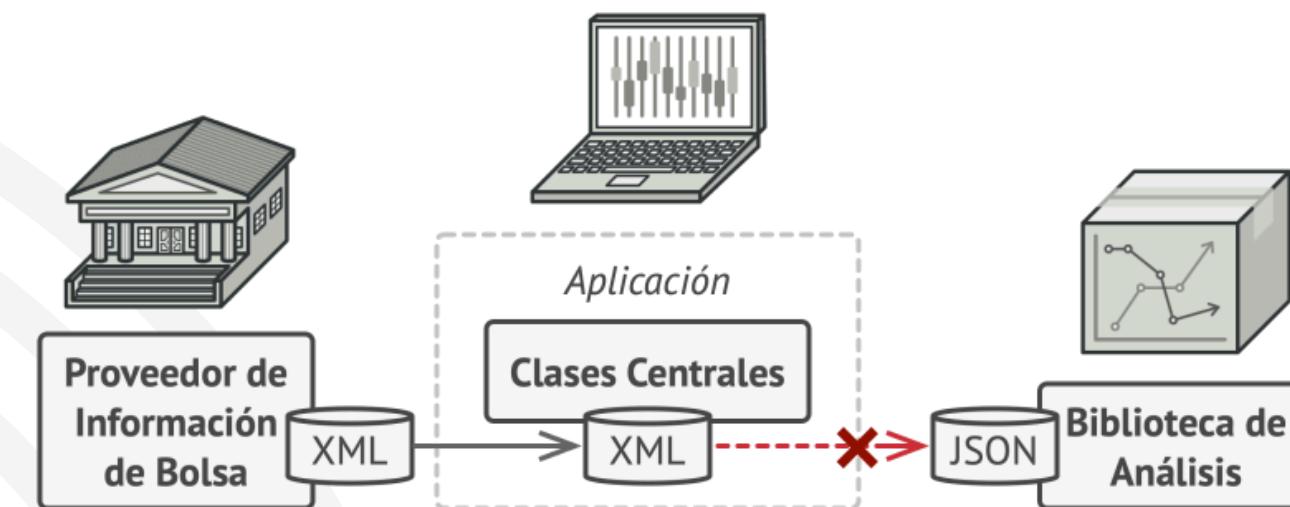
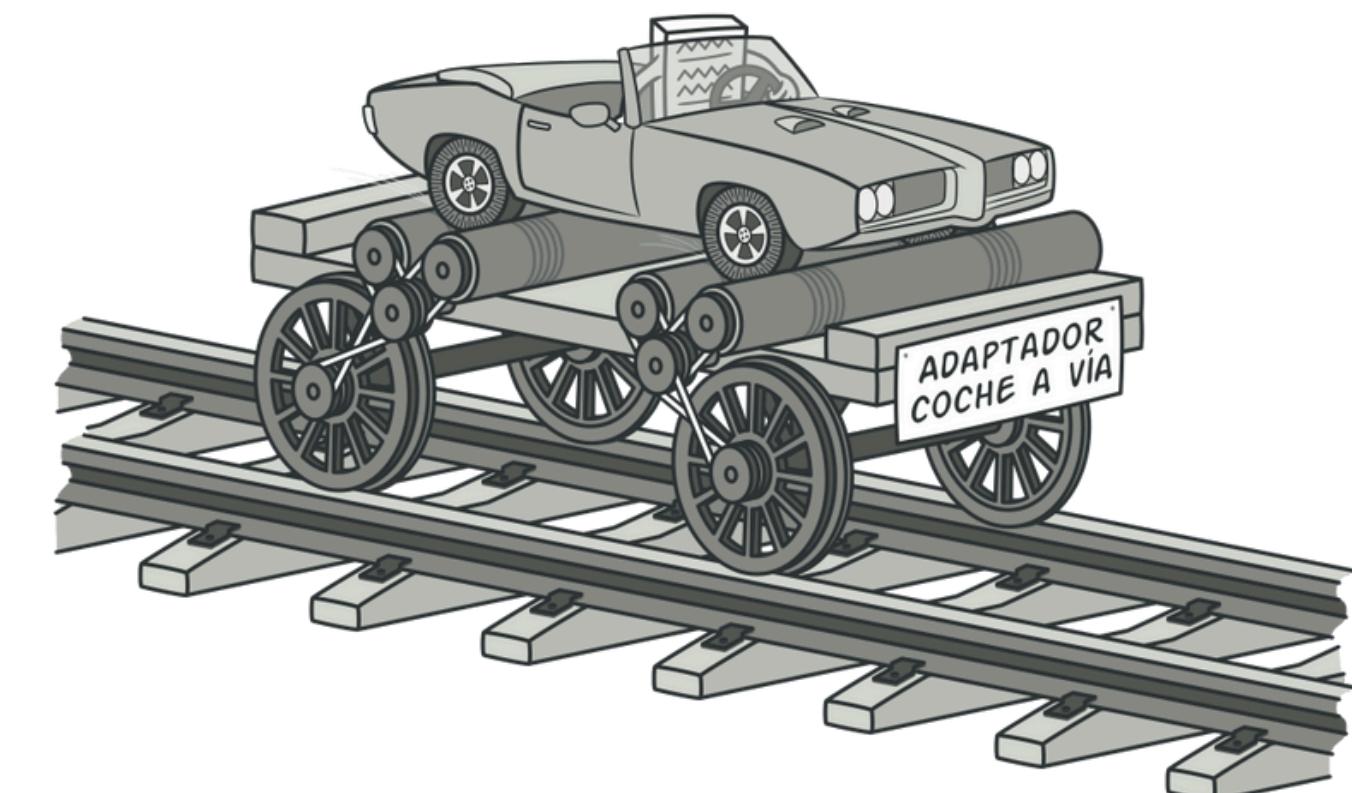
Patrones estructurales - Adaptador, Envoltorio, Wrapper



**Propósito:** Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.



**Problema:** Tu app usa datos de bolsa en XML, pero la nueva librería de análisis solo acepta JSON. No puedes modificar la librería ni romper tu código existente, por lo que necesitas un adaptador entre ambos formatos..

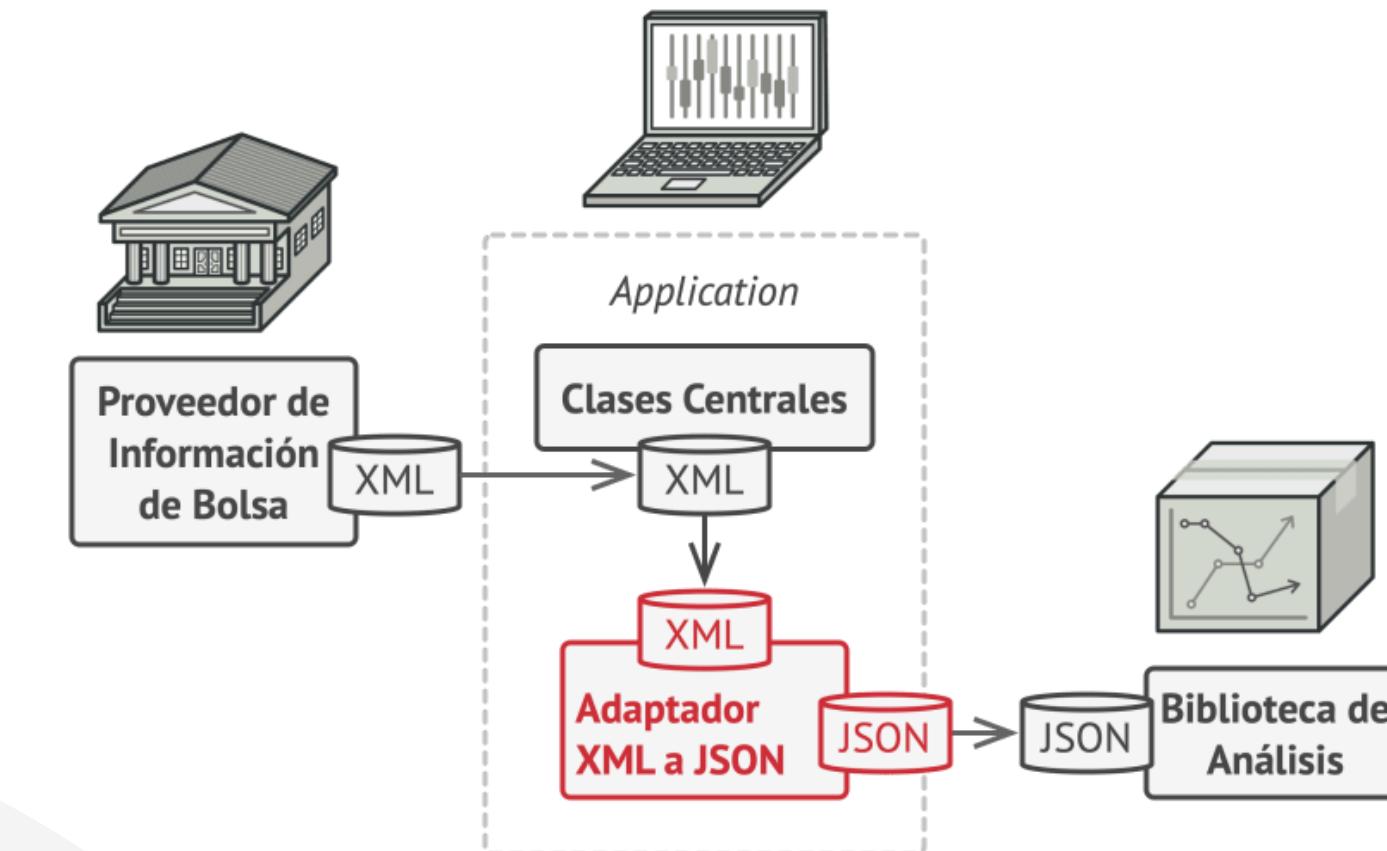


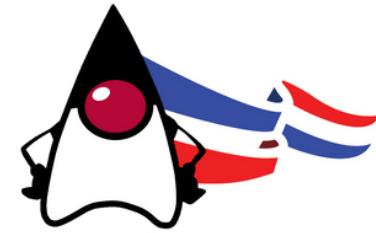
# Patrones de Diseño - Adapter

Patrones estructurales - Adaptador, Envoltorio, Wrapper



**Solución:** El patrón Adapter permite que objetos con interfaces distintas colaboren. Un adaptador envuelve a un objeto y convierte datos o llamadas sin que este lo sepa. Por ejemplo, puede transformar metros y kilómetros a pies y millas. Así se logra que un objeto que espera un formato distinto funcione sin modificarlo. El adaptador ofrece una interfaz compatible, recibe las peticiones y las traduce al formato que entiende el objeto real. Incluso puede ser bidireccional. En la app de bolsa, la solución es crear adaptadores de XML a JSON que traduzcan datos y deleguen en la librería de análisis, evitando cambios en el código existente.





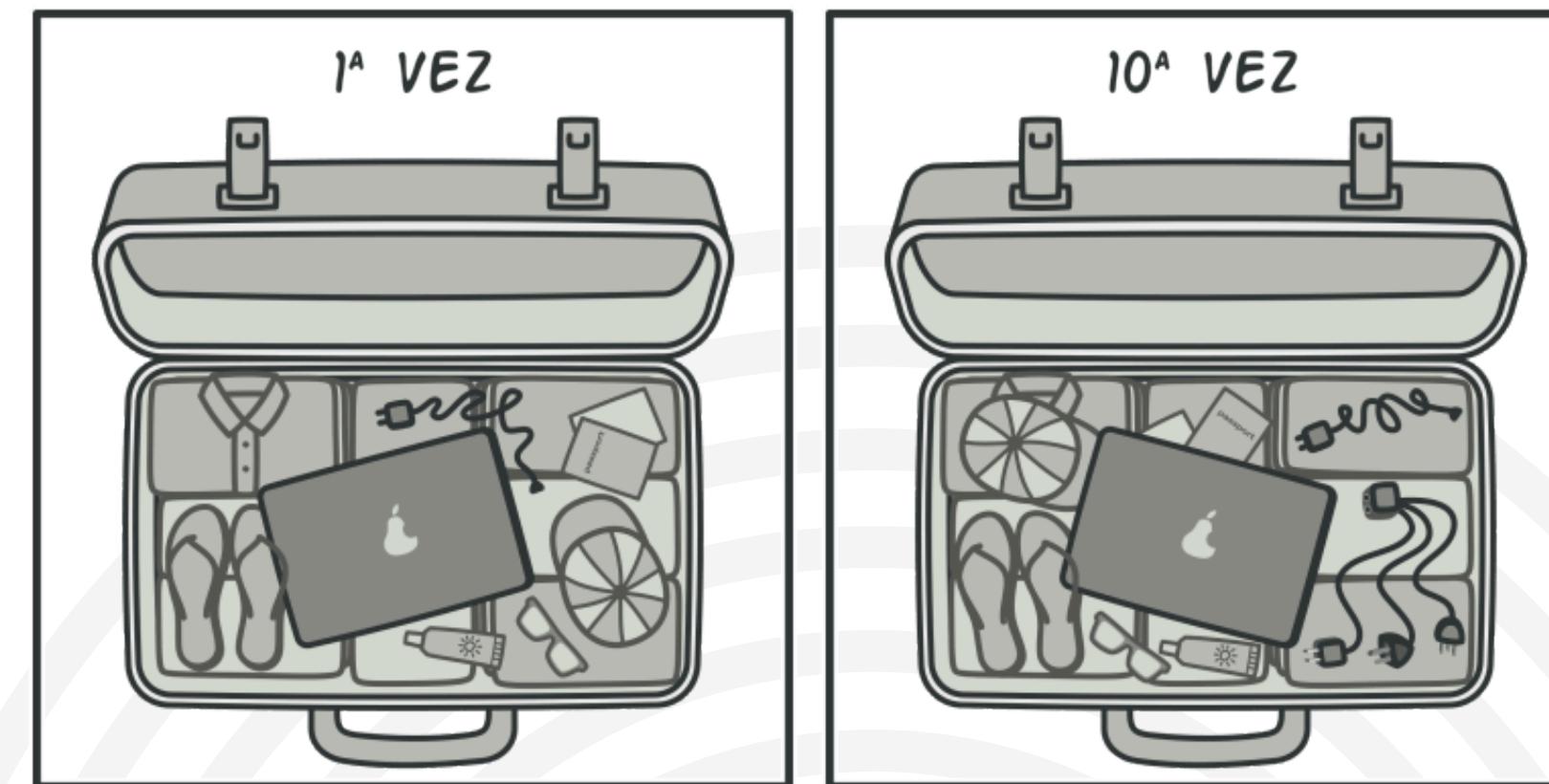
# Patrones de Diseño - Adapter

Patrones estructurales - Adaptador, Envoltorio, Wrapper



**Analogía en el mundo real:** Cuando viajas de Europa a Estados Unidos por primera vez, puede ser que te lleves una sorpresa cuando intentes cargar tu computadora portátil. Los tipos de enchufe son diferentes en cada país, por lo que un enchufe español no sirve en Estados Unidos. El problema puede solucionarse utilizando un adaptador que incluya el enchufe americano y el europeo.

VIAJAR AL EXTRANJERO



# Patrones de Diseño - Adapter

Patrones estructurales - Adaptador, Envoltorio, Wrapper



## Aplicabilidad:

- Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase..

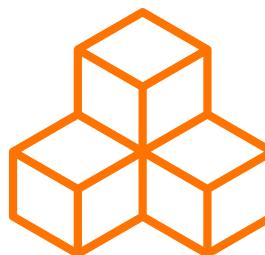


## Pros y contras:

- SRP y abierto/cerrado: separa conversión y lógica, permite nuevos adaptadores sin tocar cliente.
- Más complejidad: requiere varias clases, a veces conviene adaptar el servicio.

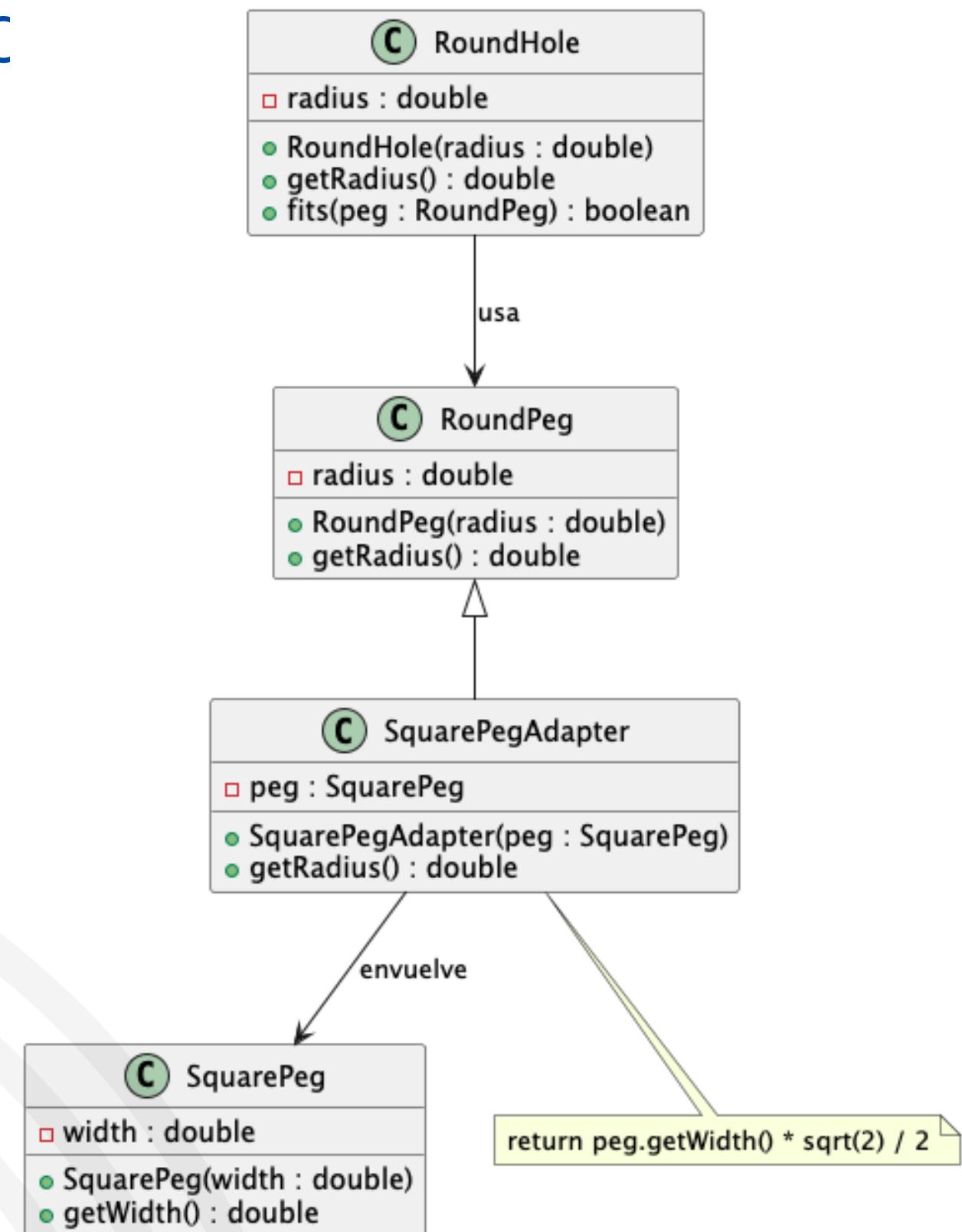
# Patrones de Diseño - Adapter

## Patrones estructurales - Adaptador, Envoltorio



- **Estructura:**
- **RoundHole** Es la lógica existente que solo sabe trabajar con piezas redondas. Su contrato es: “dame algo que tenga **getRadius()** y te digo si encaja con **fits(...)**”.
- **RoundPeg** Define lo mínimo que el cliente necesita: **getRadius()**. Cualquier cosa que luzca como un RoundPeg puede ser usada por **RoundHole**.
- **SquarePeg** Es una clase útil ya existente (o de un tercero) cuya **interfaz no coincide** con lo que el cliente espera. Expone **getWidth()**, no **getRadius()**.
- **SquarePegAdapter** demuestra:
  - **Envuelve** un **SquarePeg** (composición) y se hace pasar por un **RoundPeg** (en el ejemplo Java, heredamos de **RoundPeg** para cumplir el contrato del cliente).
  - **Traduce** la interfaz: sobrescribe **getRadius()** para devolver el radio “equivalente” de una pieza cuadrada que quepa en un círculo.
  - Así, el cliente puede llamar **hole.fits(adapter)** sin enterarse de que por dentro hay una pieza cuadrada.

Patrón Adapter: Piezas cuadradas en hoyos redondos



→ Association

→ Inheritance

→ Implementation

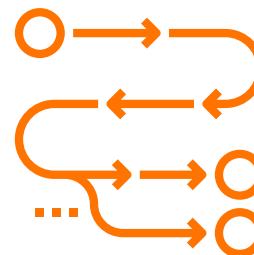
→ Dependency

→ Aggregation

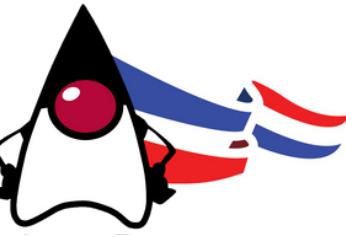
→ Composition

# Patrones de Diseño - Adapter

## Patrones estructurales - Adaptador, Envoltorio, Wrapper



- **Flujo de uso**
  - **RoundHole** usa **RoundPeg** directamente → funciona.
  - **RoundHole** no puede usar **SquarePeg** (tipos incompatibles).
  - Creamos **SquarePegAdapter(sqPeg)** → ahora sí podemos pasarlo a **fits(...)**.
  - El resultado depende de la geometría convertida (**smallAdapter** encaja, **largeAdapter** no).
- **Beneficios clave**
  - **Bajo acoplamiento**: el cliente sigue acoplado a la interfaz (**RoundPeg**), no a implementaciones concretas.
  - **OCP**: si mañana cambia la clase de servicio o añadimos otra variante incompatible, creamos otro adaptador sin tocar el cliente.
  - **SRP**: la “traducción” de interfaces vive en el adaptador, no contamina ni al cliente ni al servicio.
- **Variantes**
  - **Object Adapter** (este ejemplo): usa composición (el más común y aplicable en Java/C#/...).
  - **Class Adapter**: usa herencia múltiple (solo viable en lenguajes que la soportan, como C++), el adaptador hereda de la interfaz del cliente y del servicio al mismo tiempo.



# Patrones de Diseño - Bridge

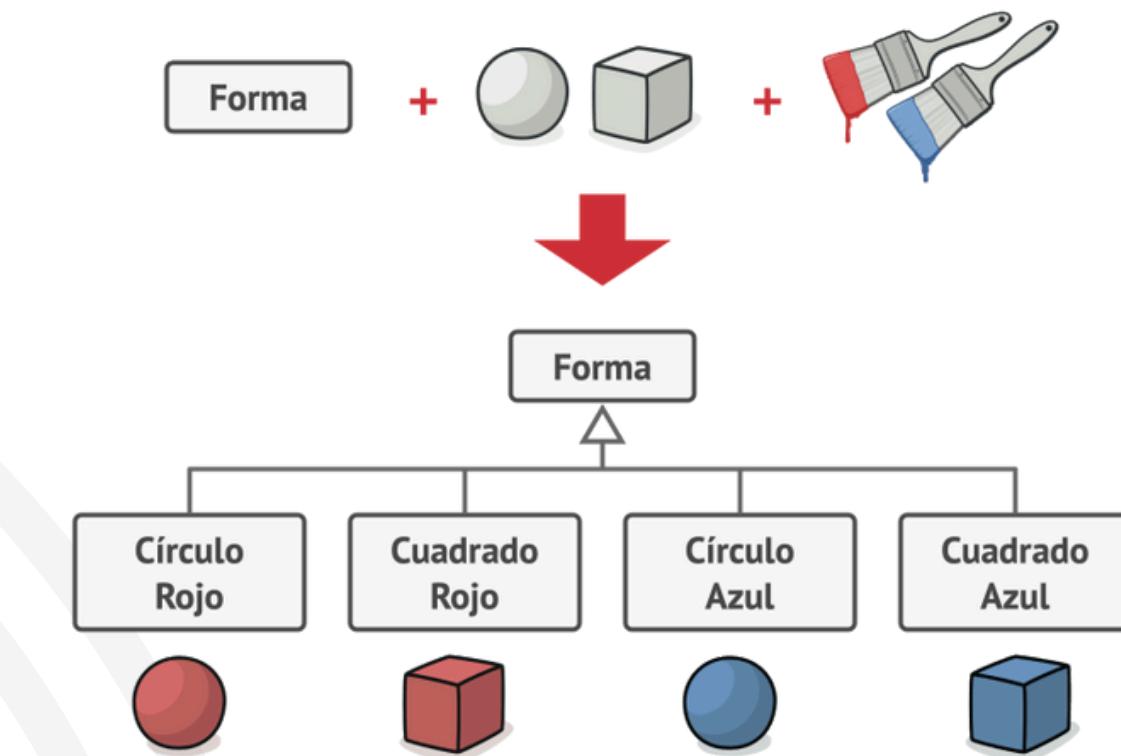
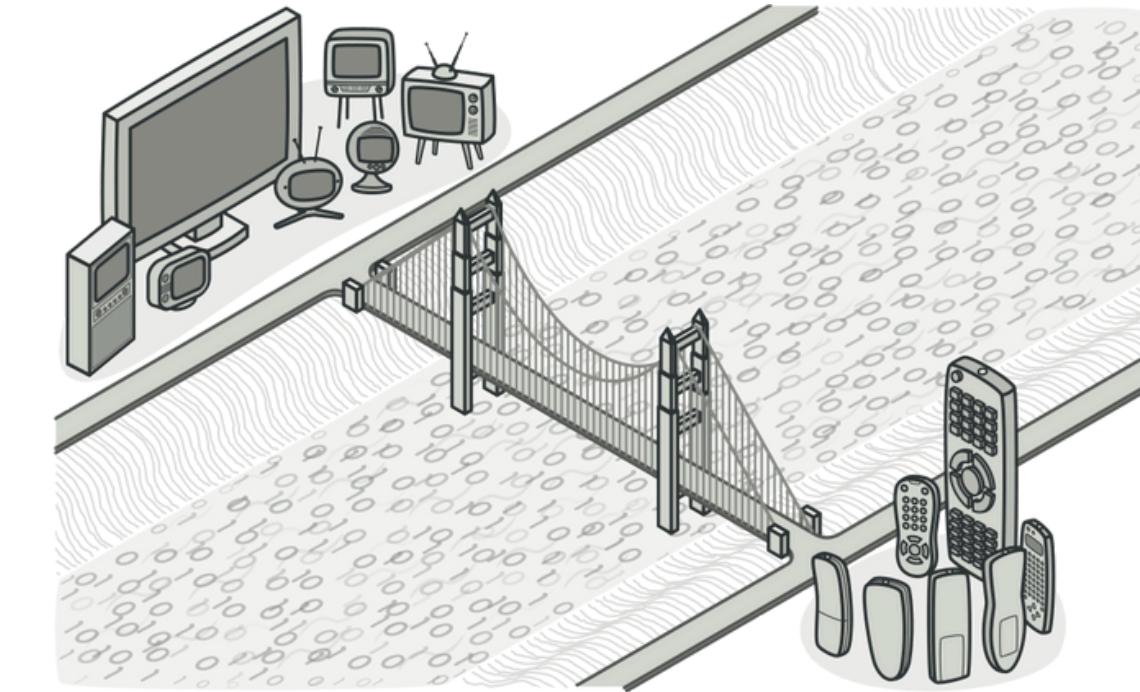
## Patrones estructurales - Puente



**Propósito:** Es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



**Problema:** El patrón Bridge evita que jerarquías crezcan exponencialmente al combinar múltiples dimensiones (como formas y colores). Separa abstracción de implementación, permitiendo extender cada una sin crear subclases por combinación.



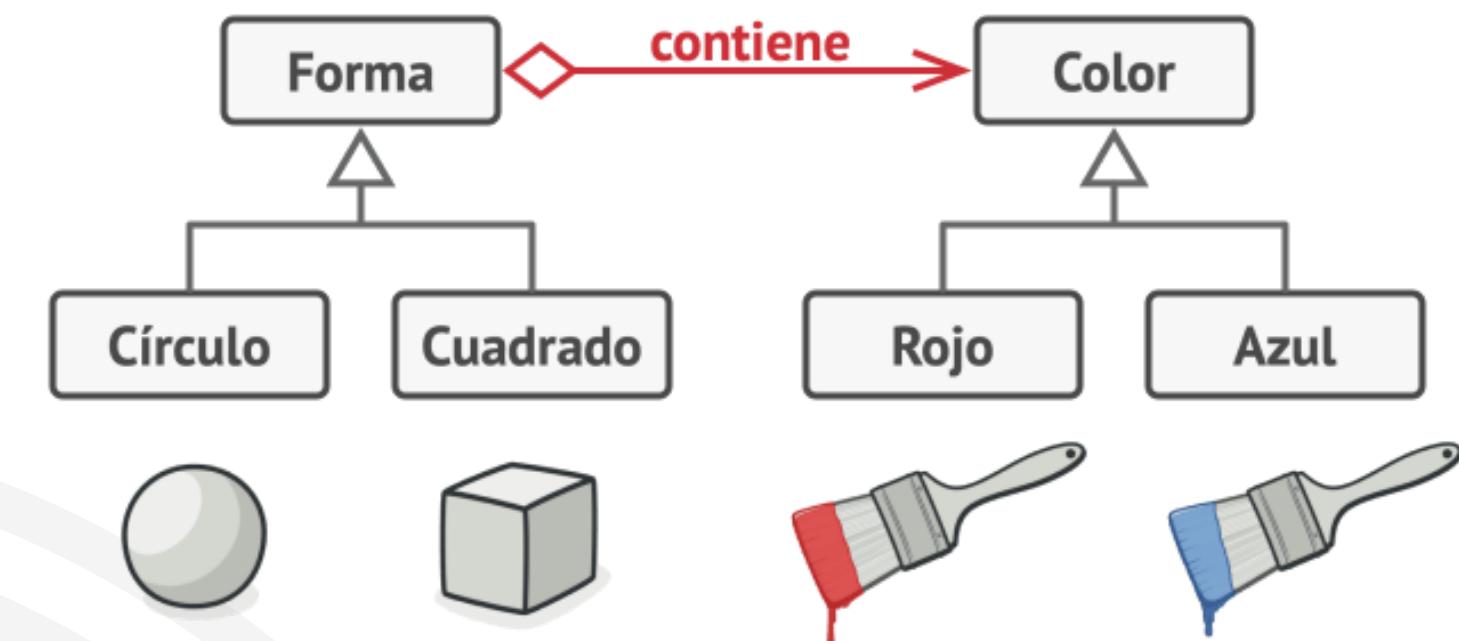
# Patrones de Diseño - Bridge

## Patrones estructurales - Puente



**Solución:** El patrón Bridge resuelve la explosión de jerarquías al separar dimensiones en clases distintas. En vez de heredar formas y colores, **Forma** referencia un objeto **Color** (Rojo, Azul) y delega en él, evitando combinaciones innecesarias. GoF llama a esto **Abstracción** (interfaz de alto nivel, como GUI) e **Implementación** (plataforma, como API).

Así, la GUI controla la apariencia y delega en la API del sistema operativo. Esto permite añadir nuevas formas o colores, o soportar nuevos sistemas (Windows, Linux, macOS) sin modificar el resto. Bridge reduce acoplamiento, facilita la extensión y mantiene el código modular.



# Patrones de Diseño - Bridge

## Patrones estructurales - Puente



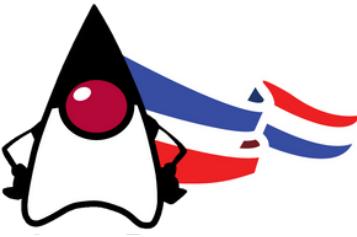
### Aplicabilidad:

- Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).
- Utiliza el patrón cuando necesites extender una clase en varias dimensiones ortogonales (independientes).
- Utiliza el patrón Bridge cuando necesites poder cambiar implementaciones durante el tiempo de ejecución.



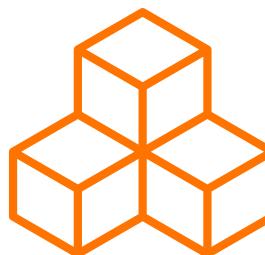
### Pros y contras:

- Bridge separa lógica y plataforma, facilita extensión y portabilidad.
- Puede añadir complejidad extra si se aplica a clases muy cohesionadas.



# Patrones de Diseño - Bridge

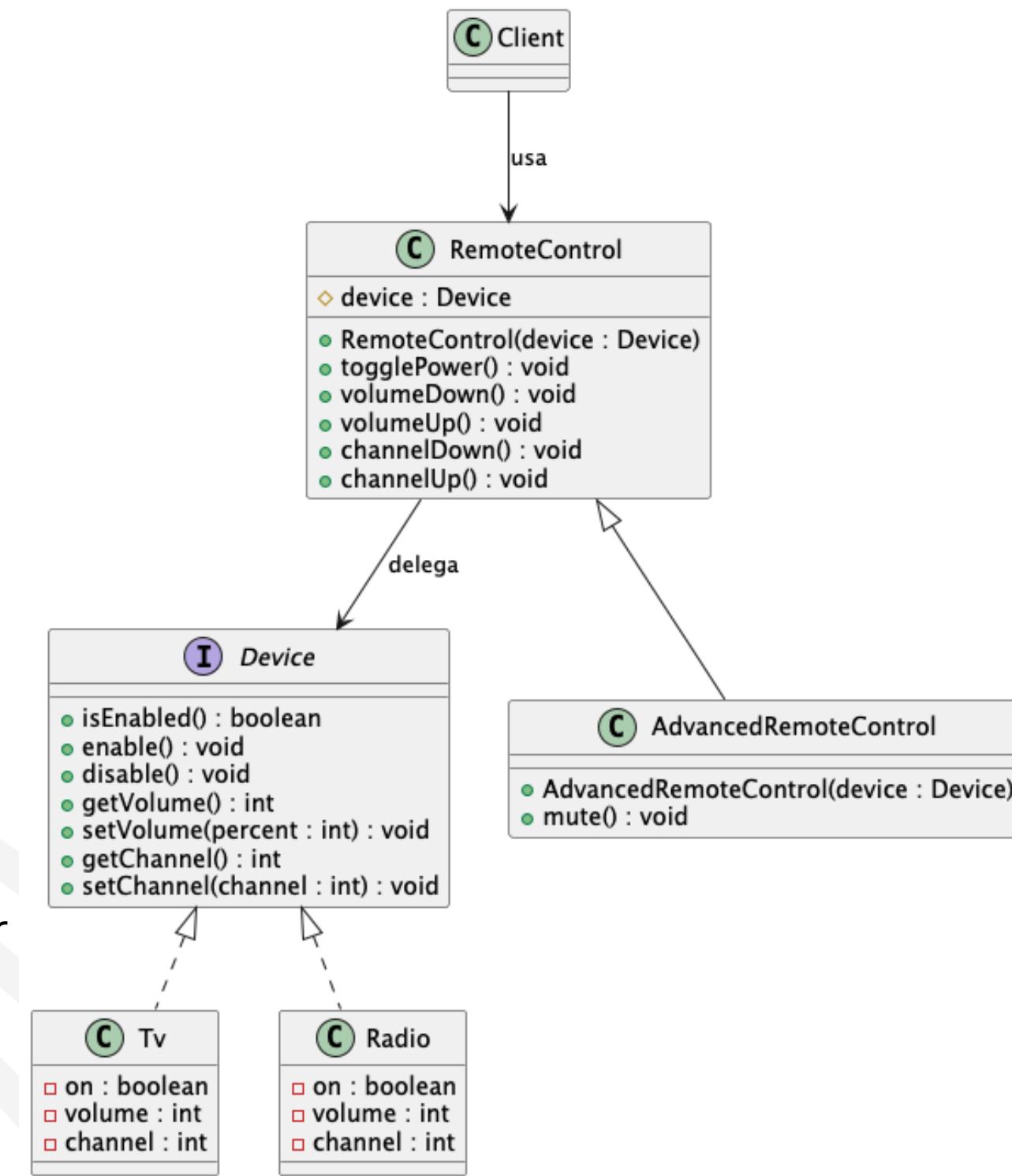
## Patrones estructurales - Puente



- **Estructura:**
- **RemoteControl** Expone la **lógica de alto nivel** (encender/apagar, subir/bajar volumen/canal) y delegá el trabajo real al objeto **Device**. No conoce detalles de plataforma.
- **AdvancedRemoteControl** Variante de la abstracción que añade funciones (ej. **mute()**), sin tocar las implementaciones..
- **Device** Interfaz mínima con **operaciones primitivas** (enable/disable/get/set), que las concreciones usarán..
- **Tv, Radio** Código específico de cada dispositivo/plataforma.
- **Cliente** Solo “ve” la **abstracción**. En tiempo de construcción inyecta el **Device** deseado. Así puedes combinar cualquier **RemoteControl** con cualquier **Device**.

**Beneficio clave:** desacopla jerarquías que cambian a ritmos distintos. Puedes crear nuevos controles o nuevos dispositivos independientemente.

Bridge Pattern: Remote Controls (Abstracción) ↔ Devices (Implementación)



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

# Patrones de Diseño - Composite

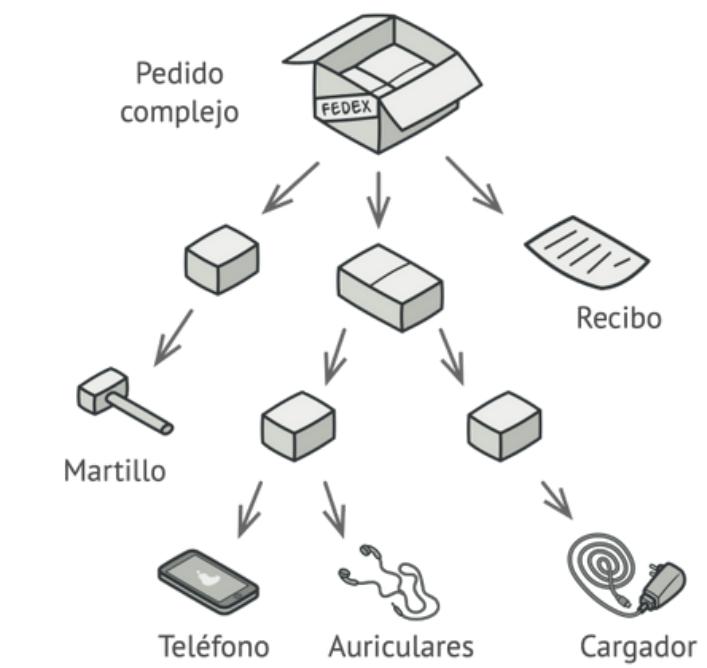
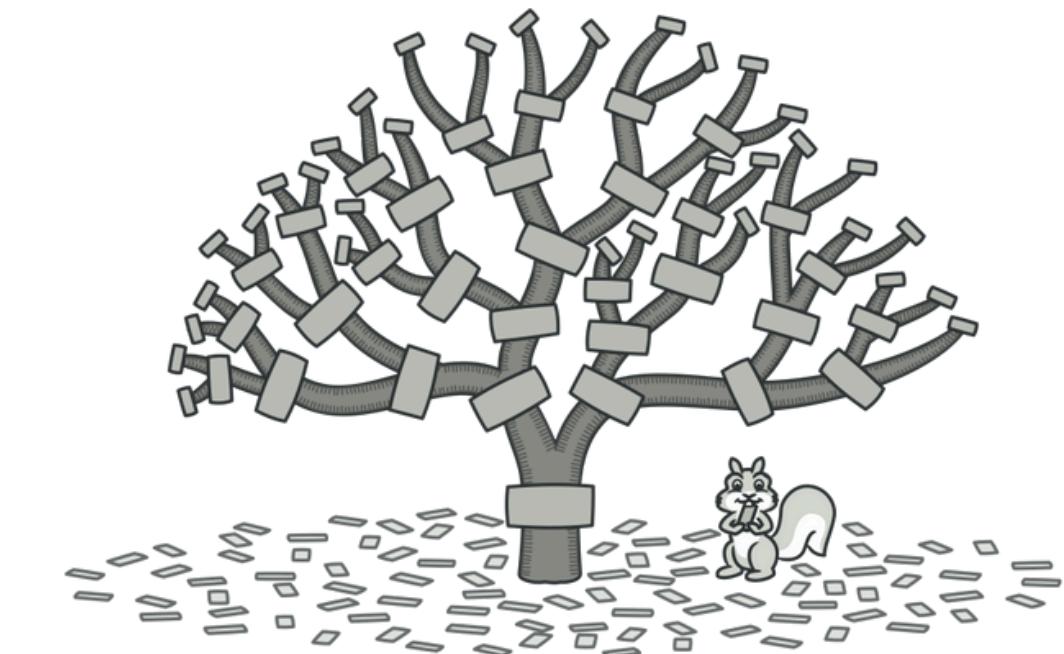
Patrones estructurales - Objeto compuesto, Object Tree



**Propósito:** Es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



**Problema:** El patrón Composite permite tratar objetos individuales y compuestos de forma uniforme. En un sistema de pedidos, productos y cajas (que contienen más productos o cajas) se organizan como un árbol para calcular totales sin lógica compleja.



# Patrones de Diseño - Composite

Patrones estructurales - Objeto compuesto, Object Tree



**Solución:** El patrón Composite propone una interfaz común para Productos y Cajas con un método de precio. Un producto devuelve su costo directo, mientras una caja suma el de sus contenidos, incluidos otros compuestos, de forma recursiva. Esto permite tratar todo como un árbol uniforme, sin distinguir entre objetos simples o complejos.

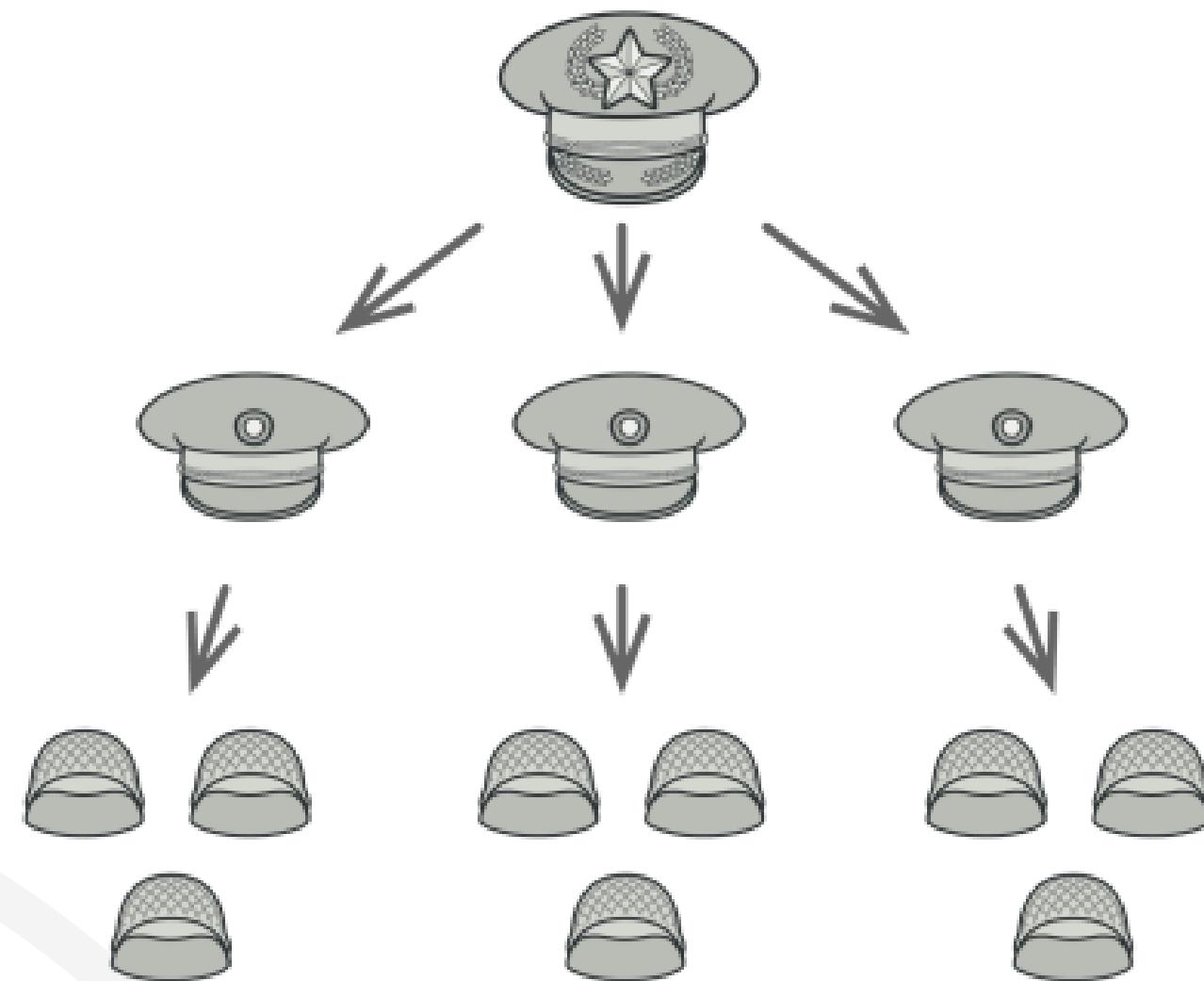


# Patrones de Diseño - Composite

Patrones estructurales - Objeto compuesto, Object Tree



**Analogía en el mundo real:** Los ejércitos de la mayoría de países se estructuran como jerarquías. Un ejército está formado por varias divisiones; una división es un grupo de brigadas y una brigada está formada por pelotones, que pueden dividirse en escuadrones. Por último, un escuadrón es un pequeño grupo de soldados reales. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben lo que hay que hacer.



# Patrones de Diseño - Composite

Patrones estructurales - Objeto compuesto, Object Tree



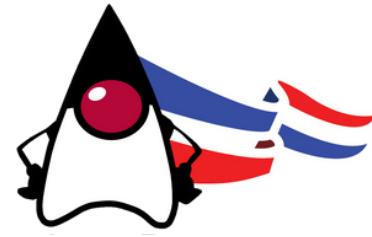
## Aplicabilidad:

- Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.
- Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.



## Pros y contras:

- Composite facilita manejar estructuras en árbol y ampliar sin modificar código.
- Difícil definir una interfaz común si las clases son muy diferentes, puede volverse confusa.



# Patrones de Diseño - Composite

## Patrones estructurales - Objeto compuesto, Object Tree

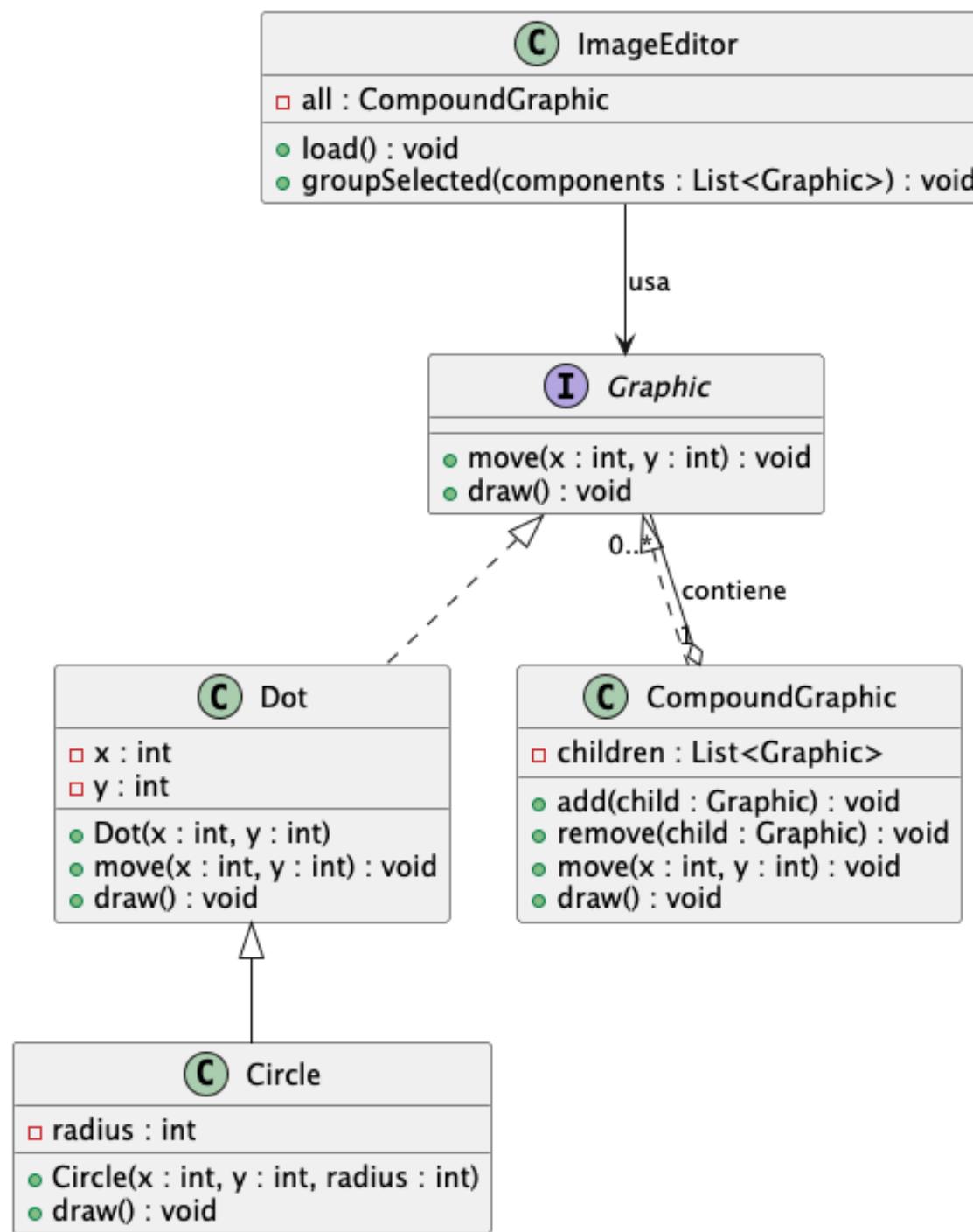


- **Estructura:**
- **Graphic** Interfaz común para hojas y compuestos. Define operaciones uniformes (**move**, **draw**) que el cliente puede invocar sin saber si el objeto es simple o compuesto.
- **Dot, Circle** Elementos **finales** del árbol. Implementan el trabajo “real”: actualizar coordenadas y dibujar su representación. **Circle** hereda de **Dot** para reutilizar **x,y**.
- **CompoundGraphic** Contiene **hijos** (cualquier **Graphic**: hojas o más compuestos). Implementa **move** y **draw** delegando recursivamente en los hijos y, si aplica, **recapitula** resultados (p. ej., calcular un bounding box).
- **ImageEditor** Trabaja **siempre** contra **Graphic**. Puede **cargar** un conjunto de formas, **agrupar** seleccionados en un nuevo compuesto y seguir usando las mismas operaciones (**move**, **draw**) sin condicionales por tipo.

### Beneficios:

- Transparencia para el cliente (trato uniforme).
- Composición recursiva de objetos (árbol).
- Abierto a extensión: nuevas hojas o compuestos implementando Graphic.

Composite Pattern: Editor de Formas (Dot, Circle, CompoundGraphic)



→ Association

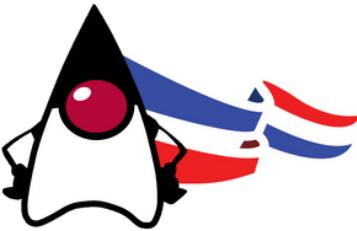
→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition



# Patrones de Diseño - Decorator

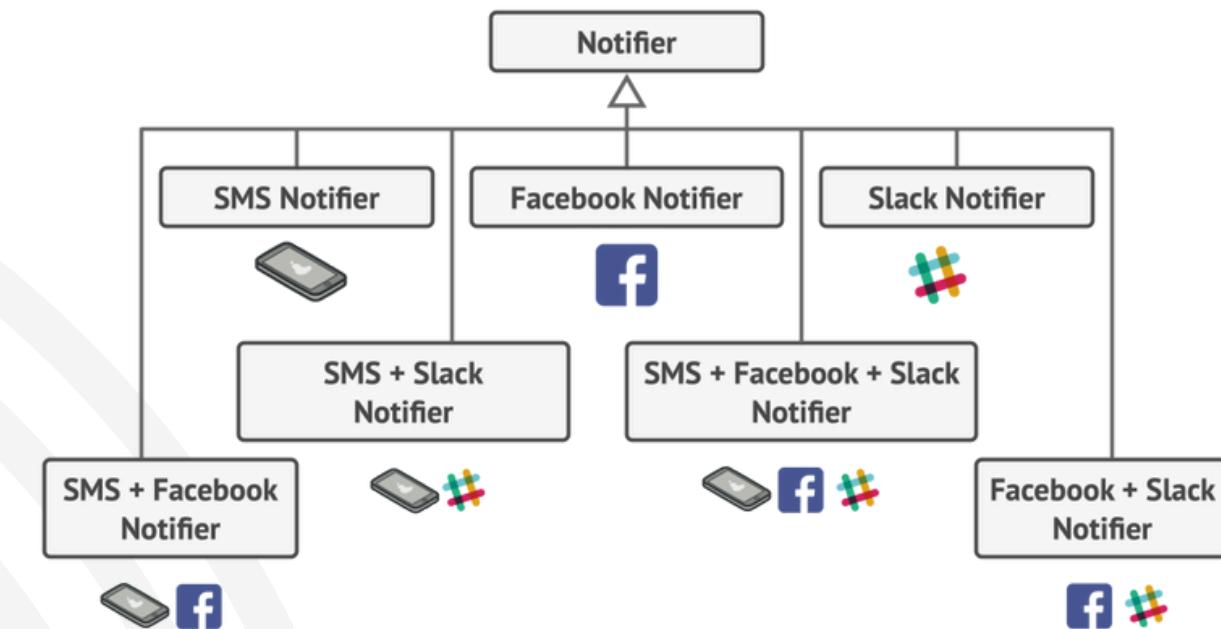
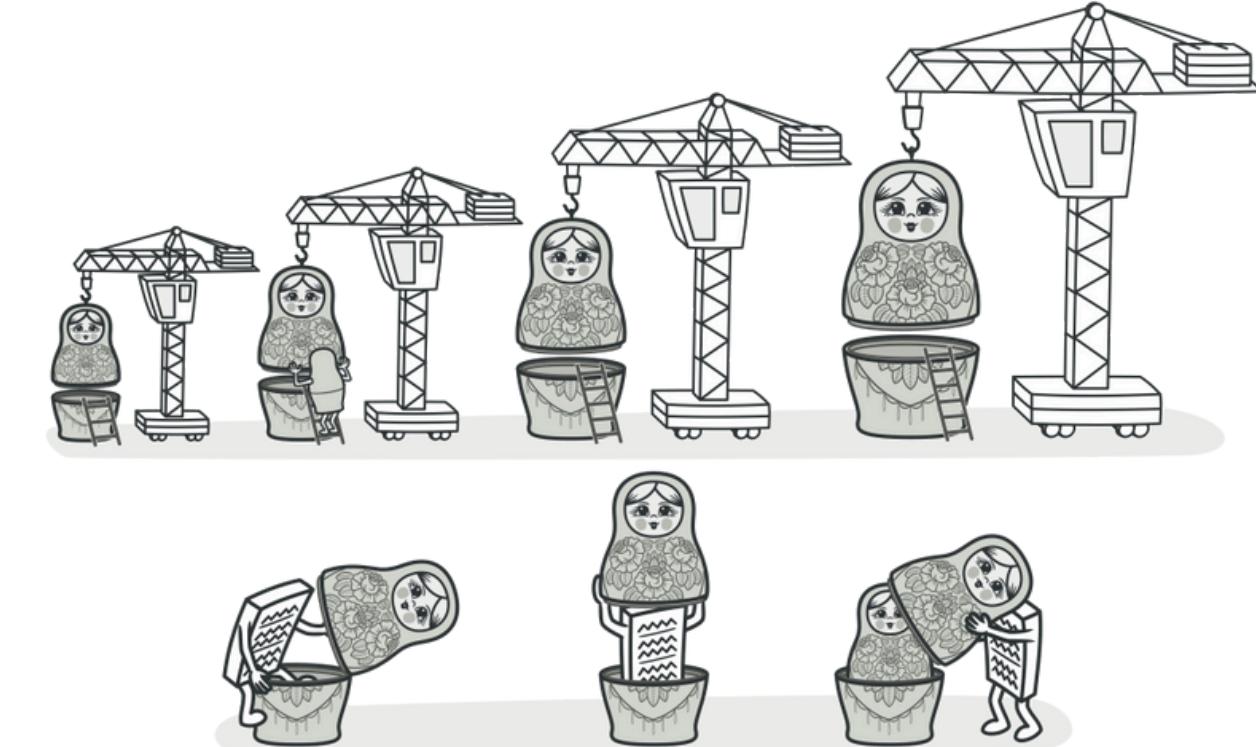
Patrones estructurales - Decorador, Envoltorio, Wrapper

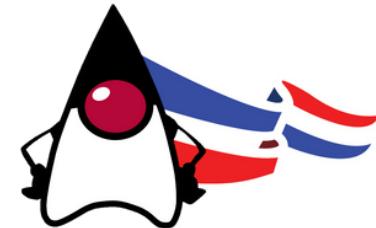


**Propósito:** Es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



**Problema:** El patrón Decorator evita la explosión de subclases en la biblioteca de notificaciones. Permite añadir dinámicamente canales (email, SMS, Slack, Facebook) envolviendo objetos y combinándolos libremente.





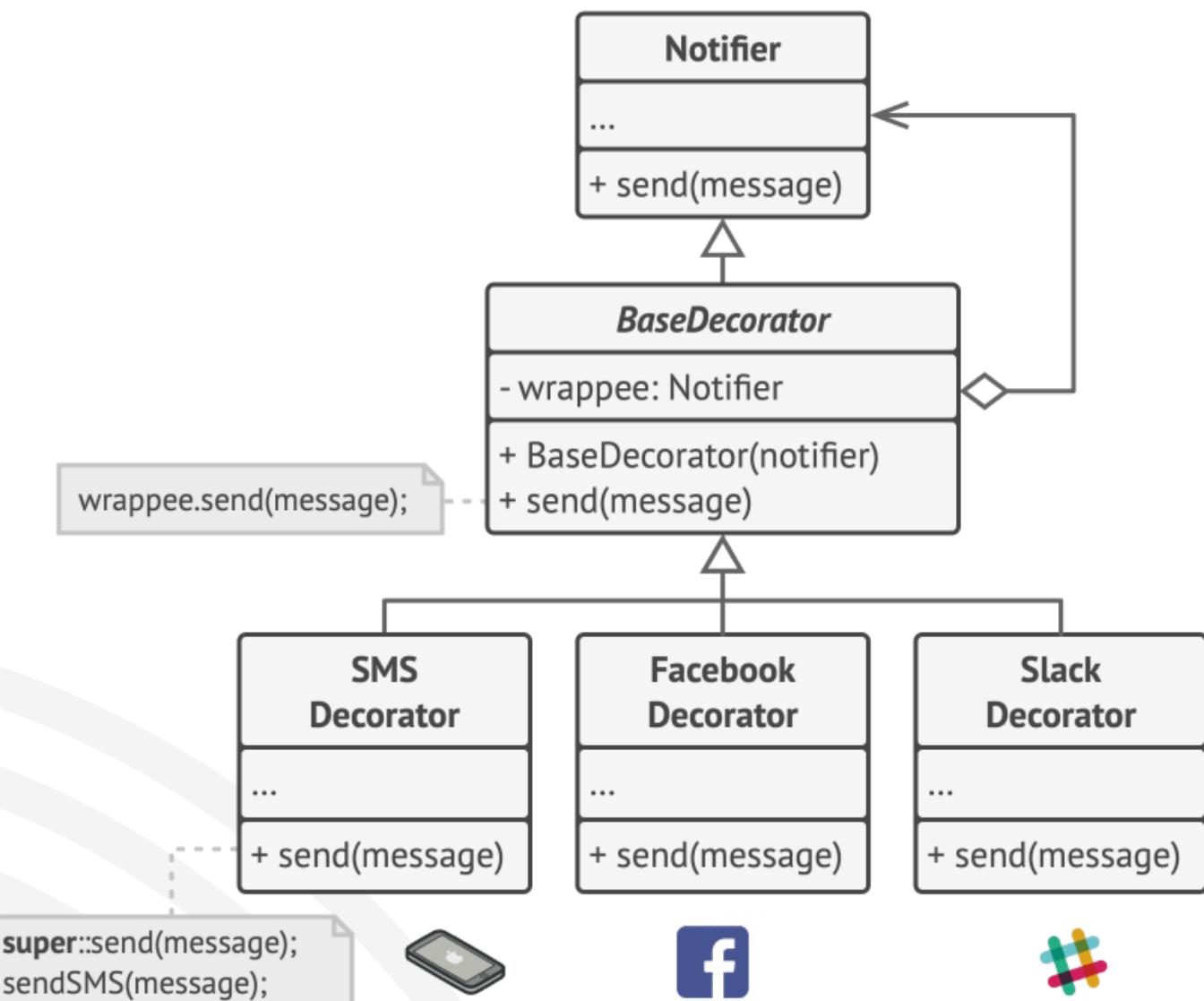
# Patrones de Diseño - Decorator

Patrones estructurales - Decorador, Envoltorio, Wrapper



**Solución:** El patrón Decorator soluciona las limitaciones de la herencia usando **composición**. Un decorador envuelve un objeto, implementa la misma interfaz y delega sus métodos, pudiendo añadir lógica antes o después. Así, el cliente no distingue entre el objeto base y el decorado. Además, puedes apilar decoradores para combinar comportamientos dinámicamente en tiempo de ejecución.

En el ejemplo de notificaciones, el **Notificador** base envía correos, mientras que decoradores añaden SMS, Slack o Facebook. El cliente construye una pila de decoradores según sus necesidades, sin crear infinitas subclases. Esta técnica es flexible, modular y mantiene el código abierto a extensiones sin modificar el núcleo.

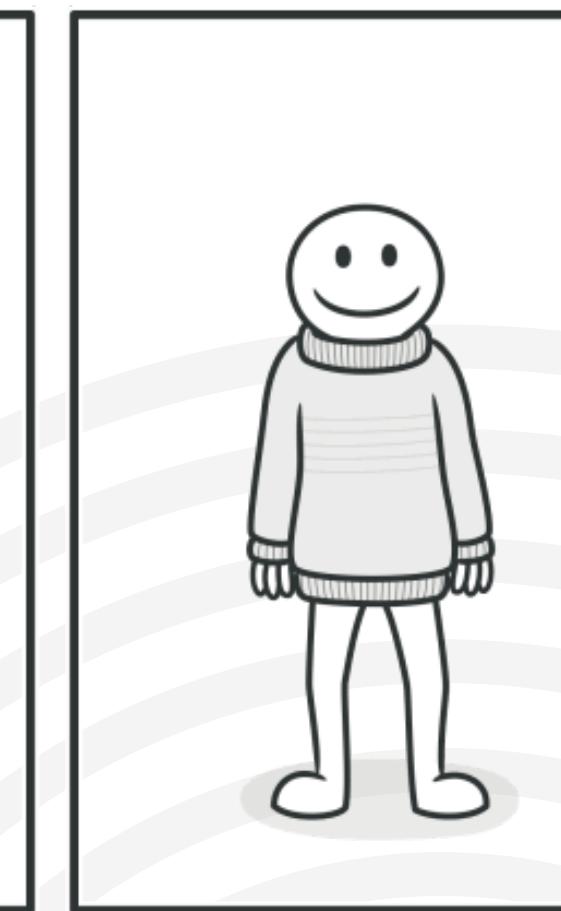


# Patrones de Diseño - Decorator

Patrones estructurales - Decorador, Envoltorio, Wrapper



**Analogía en el mundo real:** Vestir ropa es un ejemplo del uso de decoradores. Cuando tienes frío, te cubres con un suéter. Si sigues teniendo frío a pesar del suéter, puedes ponerte una chaqueta encima. Si está lloviendo, puedes ponerte un impermeable. Todas estas prendas “extienden” tu comportamiento básico pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo deseas.



# Patrones de Diseño - Decorator

Patrones estructurales - Decorador, Envoltorio, Wrapper



## Aplicabilidad:

- Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
- Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

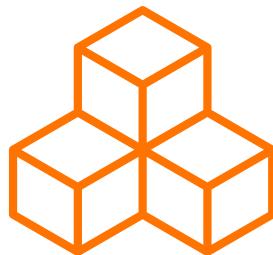


## Pros y contras:

- Decorator añade o quita responsabilidades en runtime, combina comportamientos y evita subclases.
- Difícil gestionar orden y eliminación de wrappers, configuración inicial puede ser compleja.

# Patrones de Diseño - Decorator

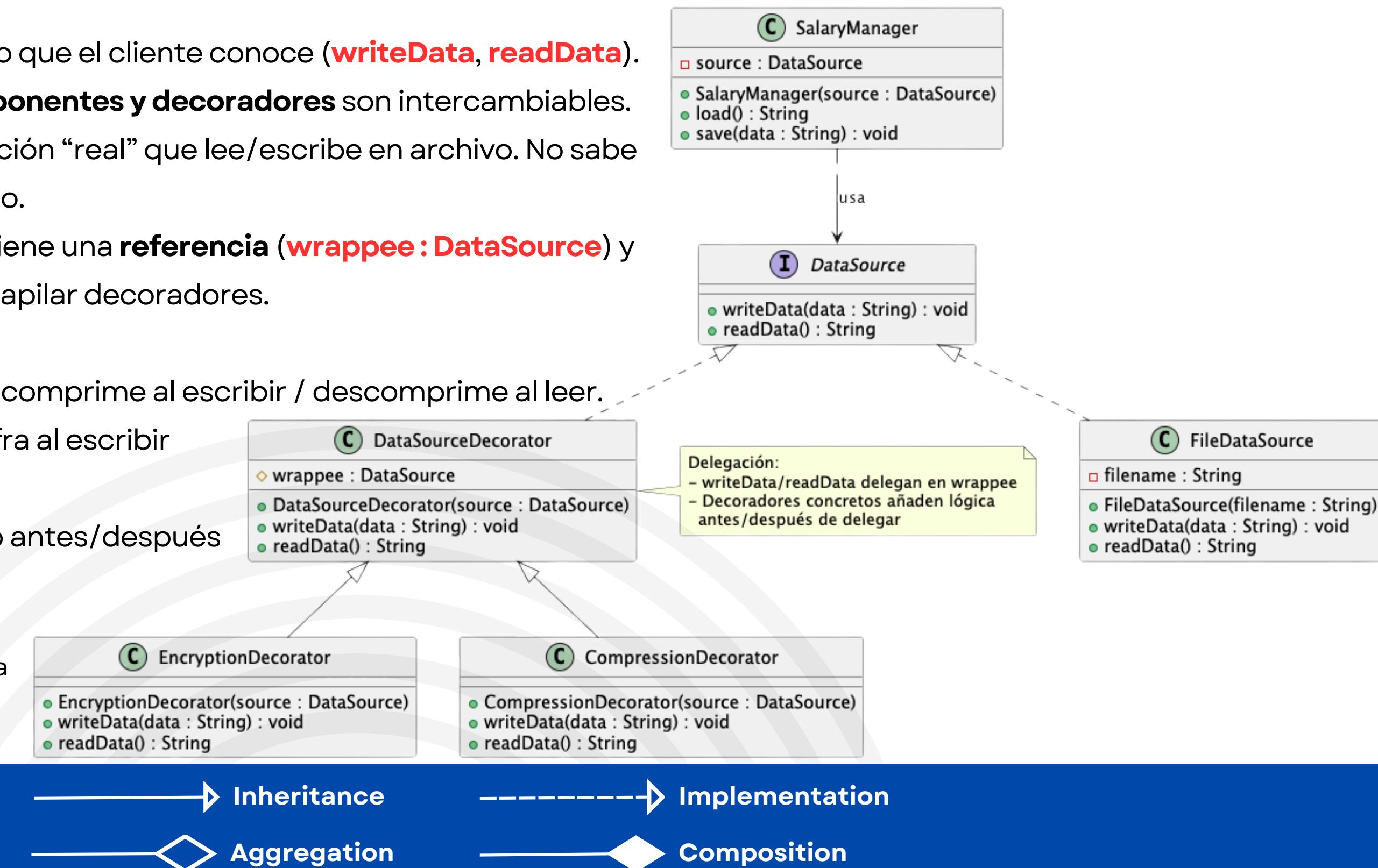
## Patrones estructurales - Decorador, Envoltorio, Wrapper



- **Estructura:**
- **DataSource** El contrato único que el cliente conoce (**writeData**, **readData**). Gracias a esta interfaz, **componentes y decoradores** son intercambiables.
- **FileDataSource** Implementación “real” que lee/escribe en archivo. No sabe nada de compresión o cifrado.
- **DataSourceDecorator** Mantiene una **referencia** (**wrappee : DataSource**) y **delegá** por defecto. Permite apilar decoradores.
- **Decoradores Concretos:**
  - **CompressionDecorator**: comprime al escribir / descomprime al leer.
  - **EncryptionDecorator**: cifra al escribir / descifra al leer.
  - Añaden comportamiento antes/después de delegar en **wrappee**.

**SalaryManager** Recibe **cualquier DataSource** ya compuesto (pila de decoradores). Trabaja igual sin conocer cómo está montada la pila.

Decorator Pattern: DataSource con Compresión + Cifrado



# Patrones de Diseño - Facade

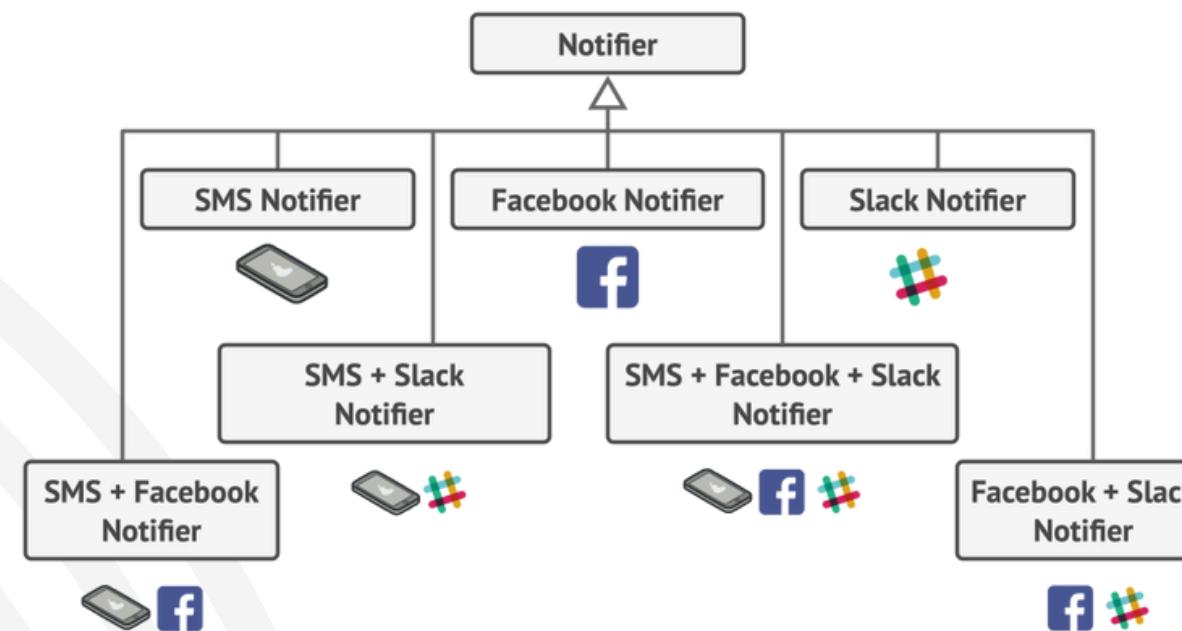
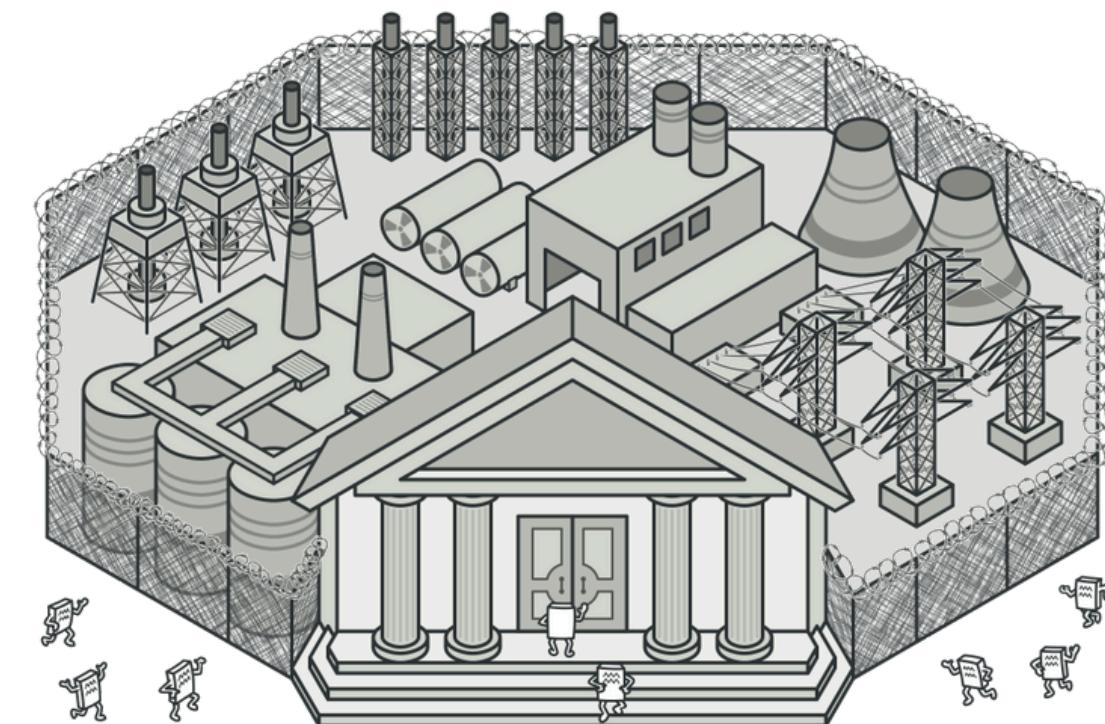
## Patrones estructurales - Fachada

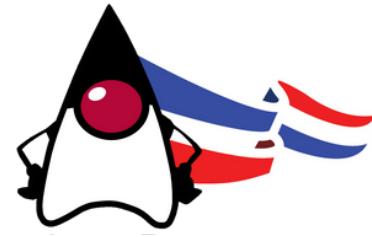


**Propósito:** Es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



**Problema:** El patrón Facade resuelve el acoplamiento con librerías complejas al proveer una interfaz simple y unificada. Así el cliente usa un punto de acceso claro sin preocuparse por dependencias ni el orden de ejecución.



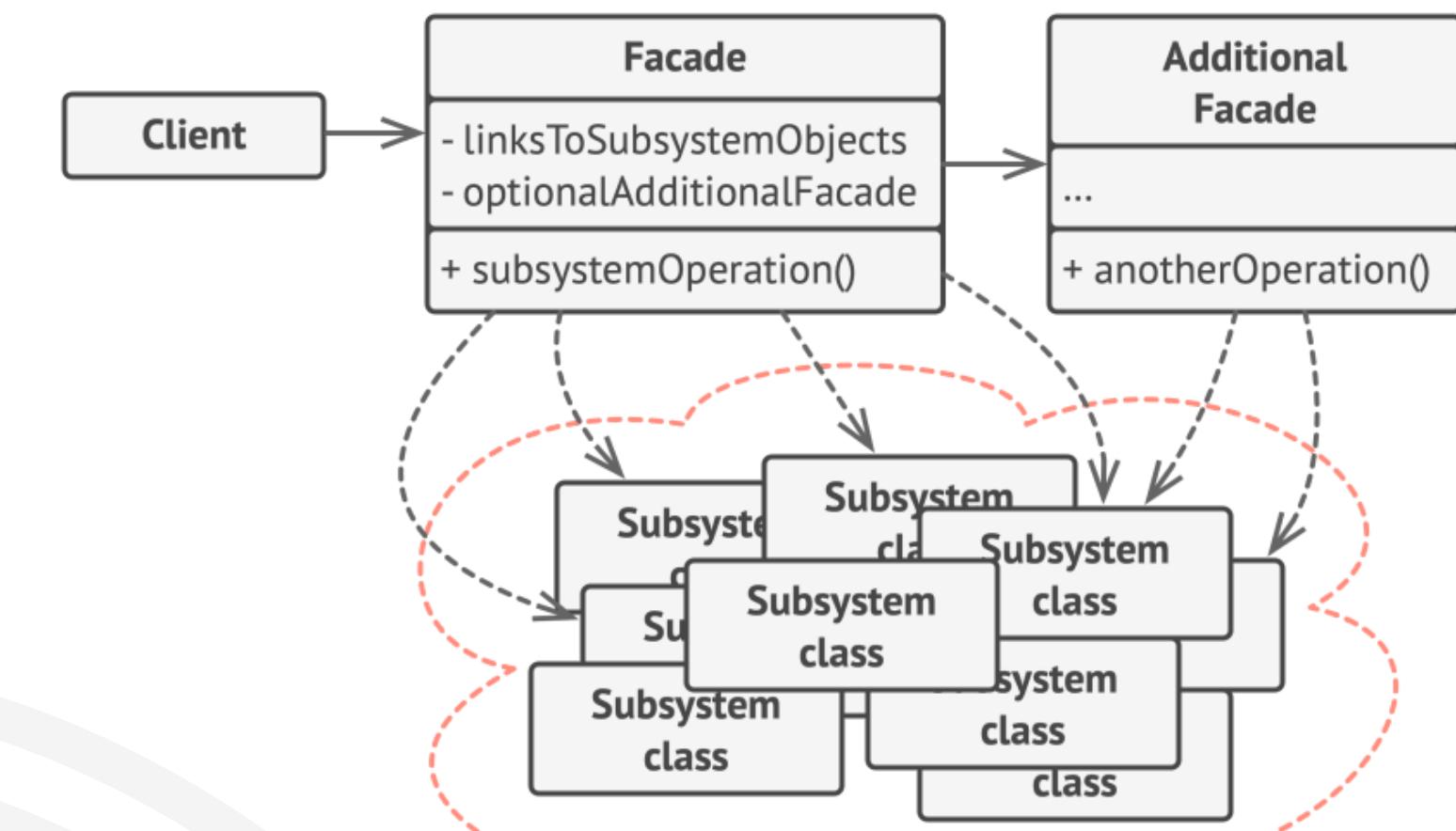


# Patrones de Diseño - Facade

## Patrones estructurales - Fachada



**Solución:** El patrón Facade ofrece una interfaz simple sobre un subsistema complejo, exponiendo solo lo esencial al cliente. Es útil al integrar librerías grandes de las que solo necesitas pocas funciones. Por ejemplo, una app que sube videos puede envolver una biblioteca profesional de conversión en una clase con un único método **codificar(archivo, formato)**. De este modo, se ocultan los detalles internos, el cliente usa una API clara y la aplicación queda desacoplada de la complejidad del subsistema.

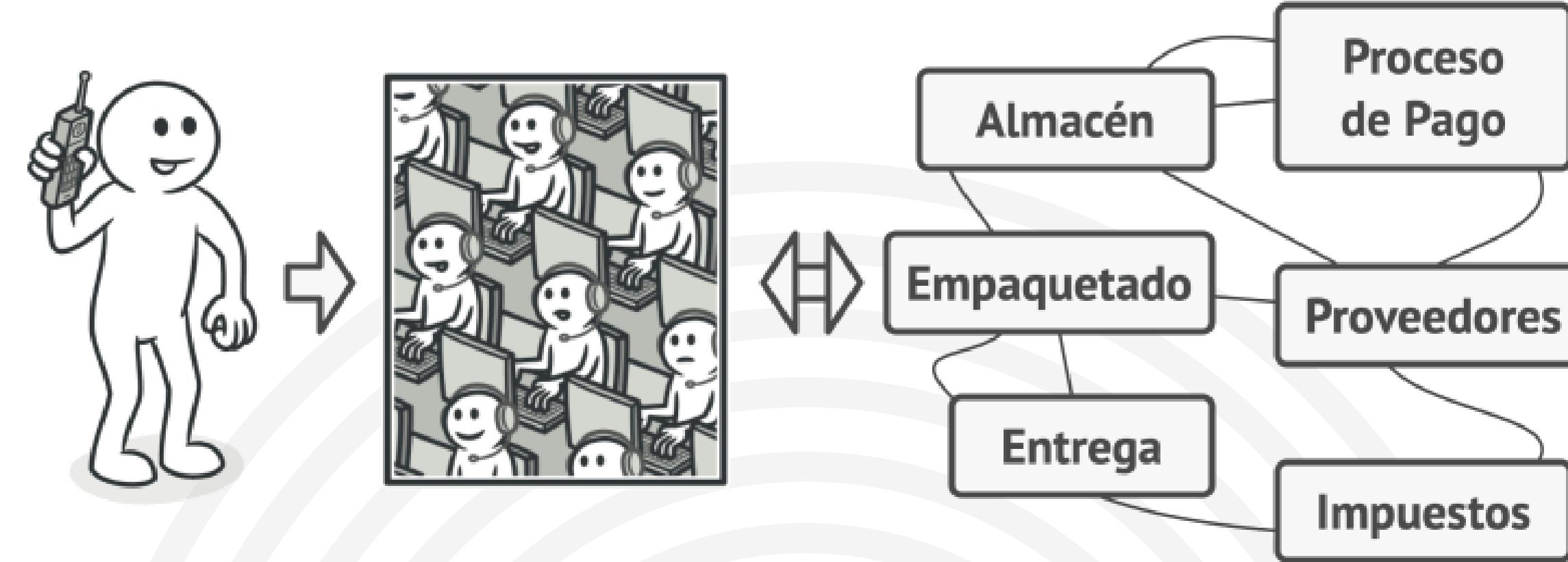


# Patrones de Diseño - Facade

## Patrones estructurales - Fachada



**Analogía en el mundo real:** Cuando llamas a una tienda para hacer un pedido por teléfono, un operador es tu fachada a todos los servicios y departamentos de la tienda. El operador te proporciona una sencilla interfaz de voz al sistema de pedidos, pasarelas de pago y varios servicios de entrega.



# Patrones de Diseño - Facade

## Patrones estructurales - Fachada



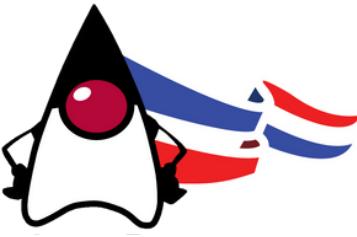
### Aplicabilidad:

- Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
- Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.



### Pros y contras:

- Puedes aislar tu código de la complejidad de un subsistema.
- Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.



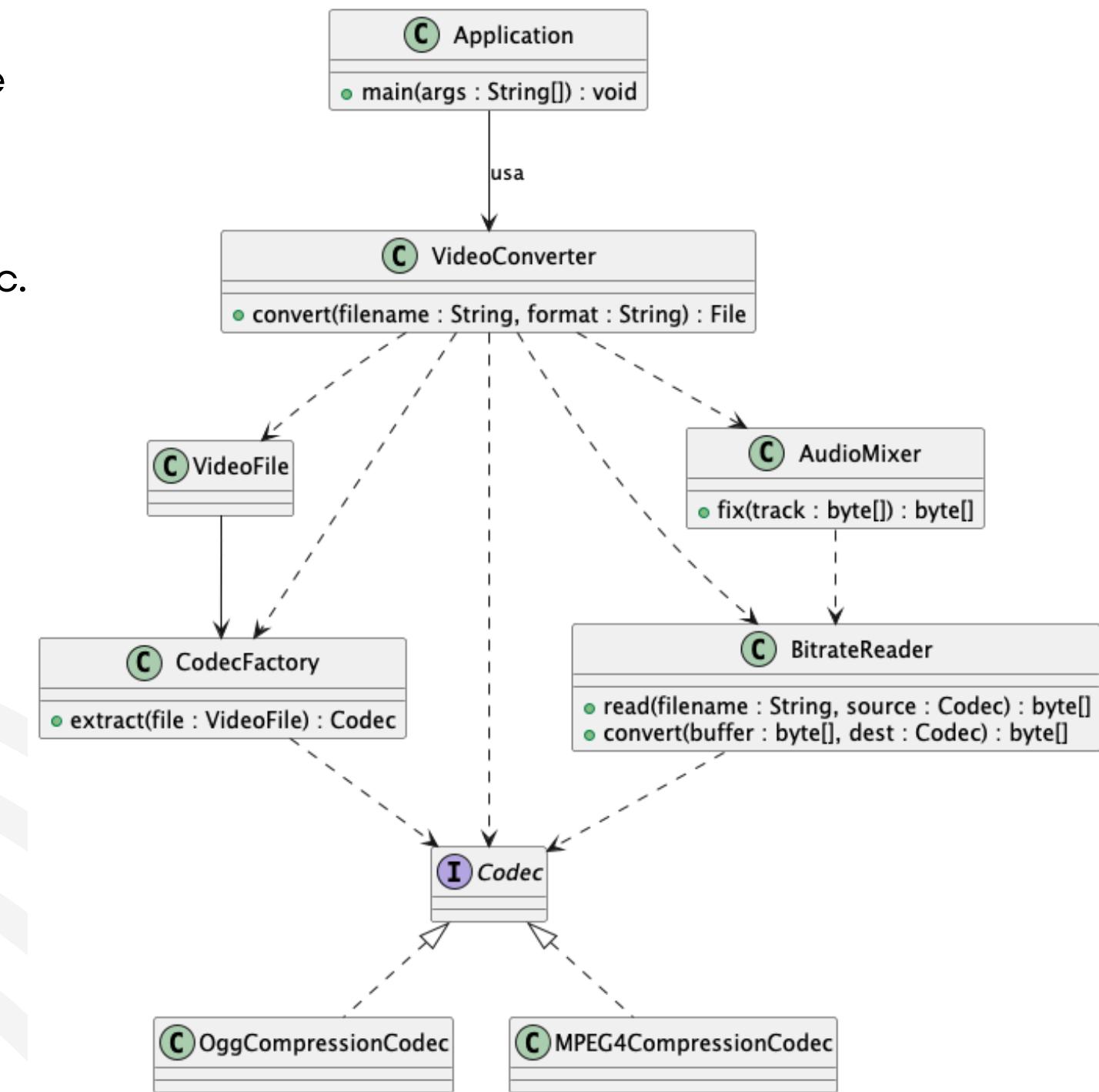
# Patrones de Diseño - Facade

## Patrones estructurales - Fachada



- **Estructura:**
- **Application** Necesita una operación de alto nivel: “**convertir video**”. No quiere lidiar con detalles del framework.
- **Subsistema complejo** Varias clases con dependencias y orden específico:  
**VideoFile, CodecFactory, Codec** (y variantes), **BitrateReader, AudioMixer**, etc.  
El cliente **no** debe conocerlas ni **orquestarlas**.
- **VideoConverter** Expone una **interfaz simple (convert(filename, format))** y **encapsula**: selección de códec, lectura/conversión de bitrate, mezcla de audio, etc. Si mañana cambia el framework, **solo** se toca la fachada.
- **Ventajas**
  - **Bajo acoplamiento** del cliente con el framework.
  - **Punto único** para aislar cambios/evolución de dependencias.
  - Interfaz **amigable** y coherente.
- **Variantes**
  - **Varias fachadas** si hay dominios distintos (p. ej., **ImageConverterFacade**, **AudioConverterFacade**) para no “sobre-cargar” una sola fachada.

**Facade Pattern: VideoConverter oculta el framework complejo**



→ Association

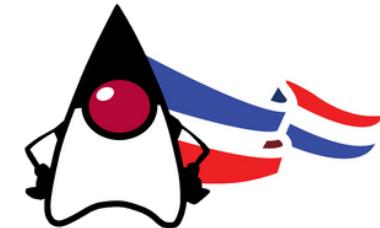
→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition



# Patrones de Diseño - Flyweight

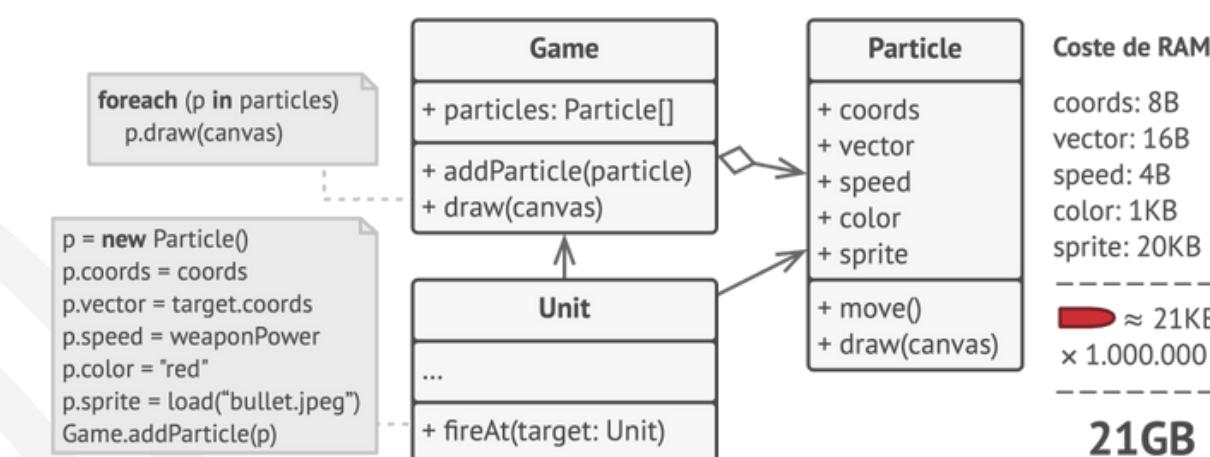
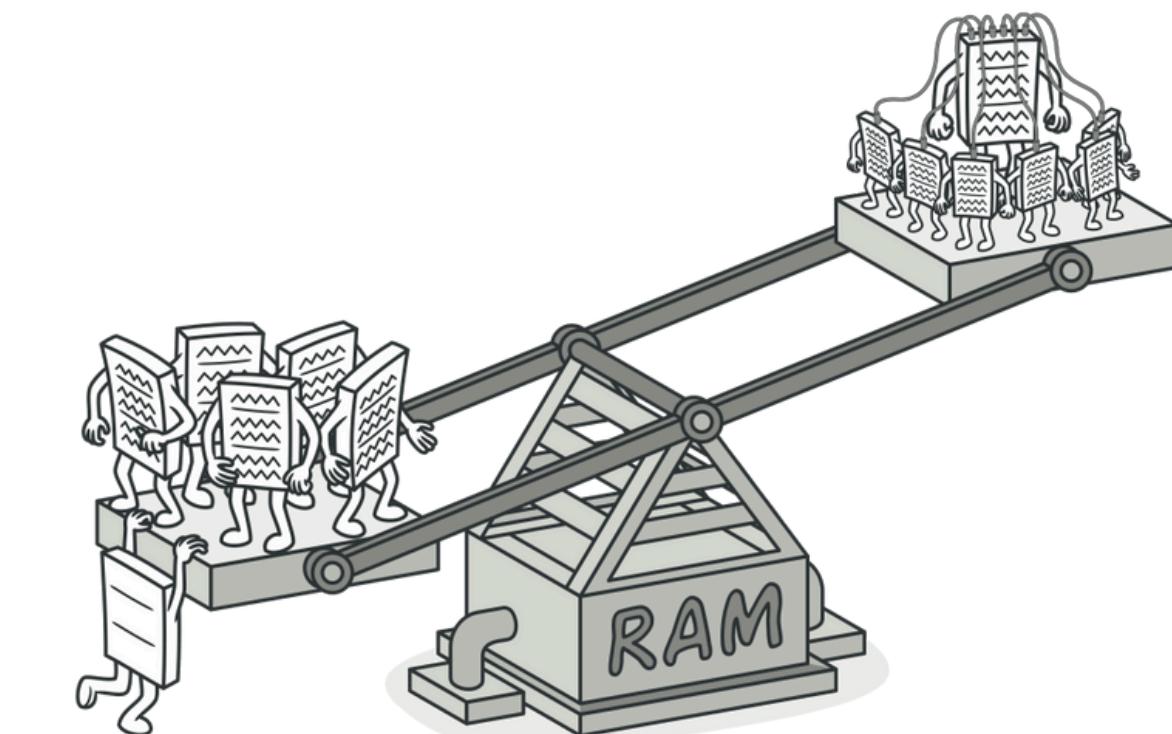
Patrones estructurales - Peso mosca, Peso ligero, Cache

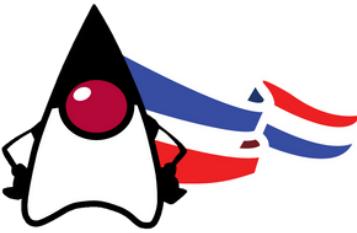


**Propósito:** Es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



**Problema:** El sistema de partículas del juego consumía demasiada RAM porque cada bala, misil o metralla se representaba con objetos pesados. Al aumentar el número en pantalla, la memoria se agotaba y el juego fallaba.





# Patrones de Diseño - Flyweight

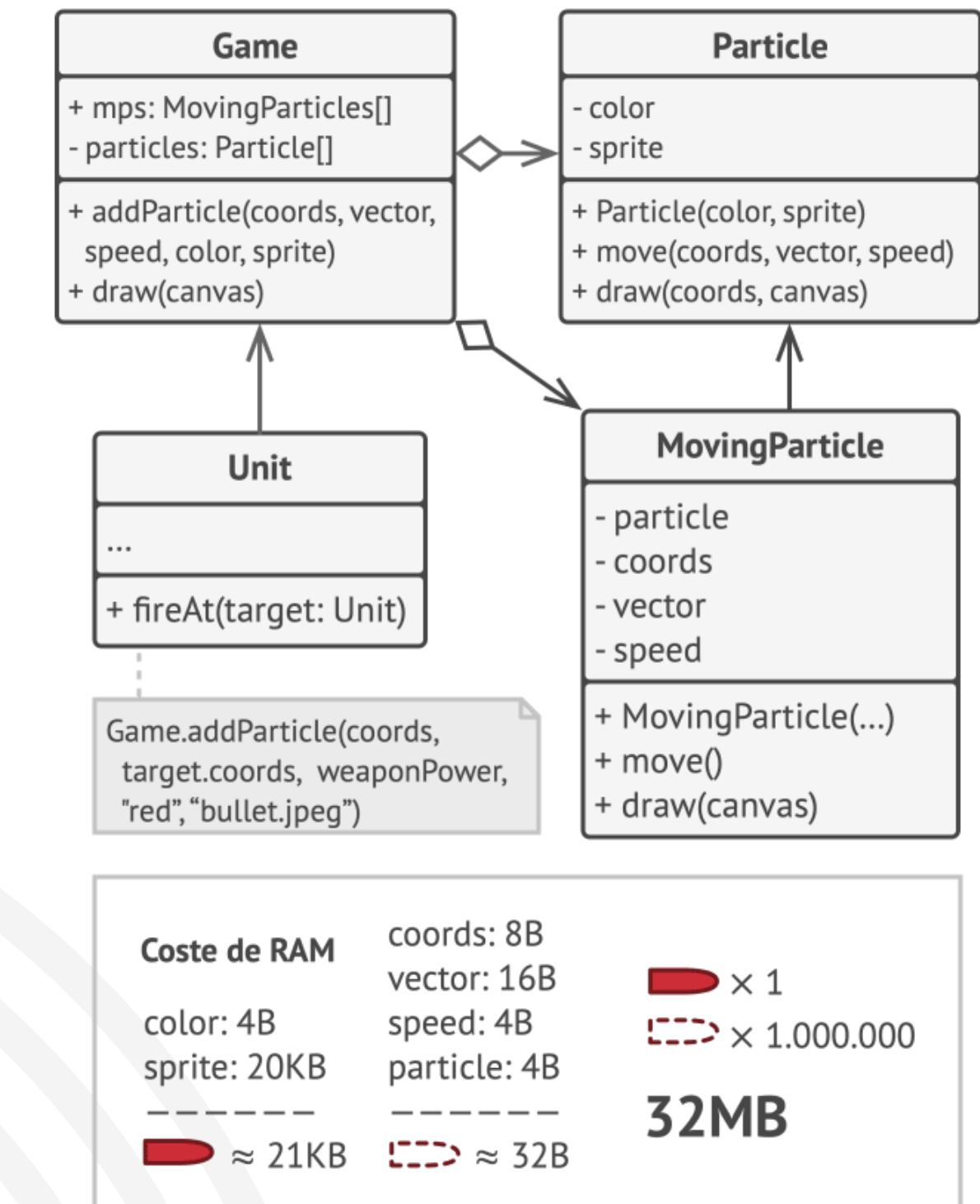
Patrones estructurales - Peso mosca, Peso ligero, Cache



**Solución:** El patrón Flyweight optimiza el uso de memoria separando el estado intrínseco (constante, ej. color, sprite) del extrínseco (cambiable, ej. posición, velocidad).

Así, solo se crean unos pocos objetos pesados (bala, misil, metralla) reutilizables en múltiples contextos. El estado extrínseco se mueve a una clase contenedora o de contexto que lo gestiona junto a la referencia al flyweight, reduciendo drásticamente el consumo.

Los flyweights deben ser inmutables para evitar inconsistencias. Para administrarlos, se recomienda usar una **fábrica flyweight** que cree y devuelva instancias compartidas según el estado intrínseco requerido.



# Patrones de Diseño - Flyweight

Patrones estructurales - Peso mosca, Peso ligero, Cache



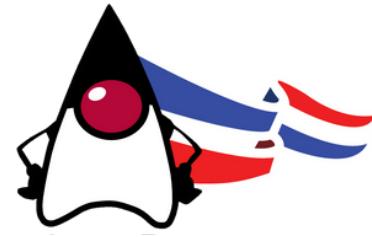
## Aplicabilidad:

- Utiliza el patrón Flyweight únicamente cuando tu programa deba soportar una enorme cantidad de objetos que apenas quepan en la RAM disponible.



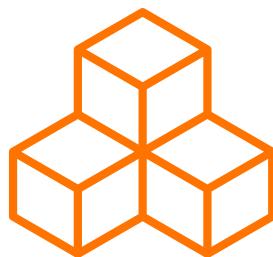
## Pros y contras:

- Puedes ahorrar mucha RAM, siempre que tu programa tenga toneladas de objetos similares.
- Flyweight ahorra RAM pero aumenta uso de CPU al recalcular contexto.
- El código se complica y es difícil de entender para nuevos desarrolladores.



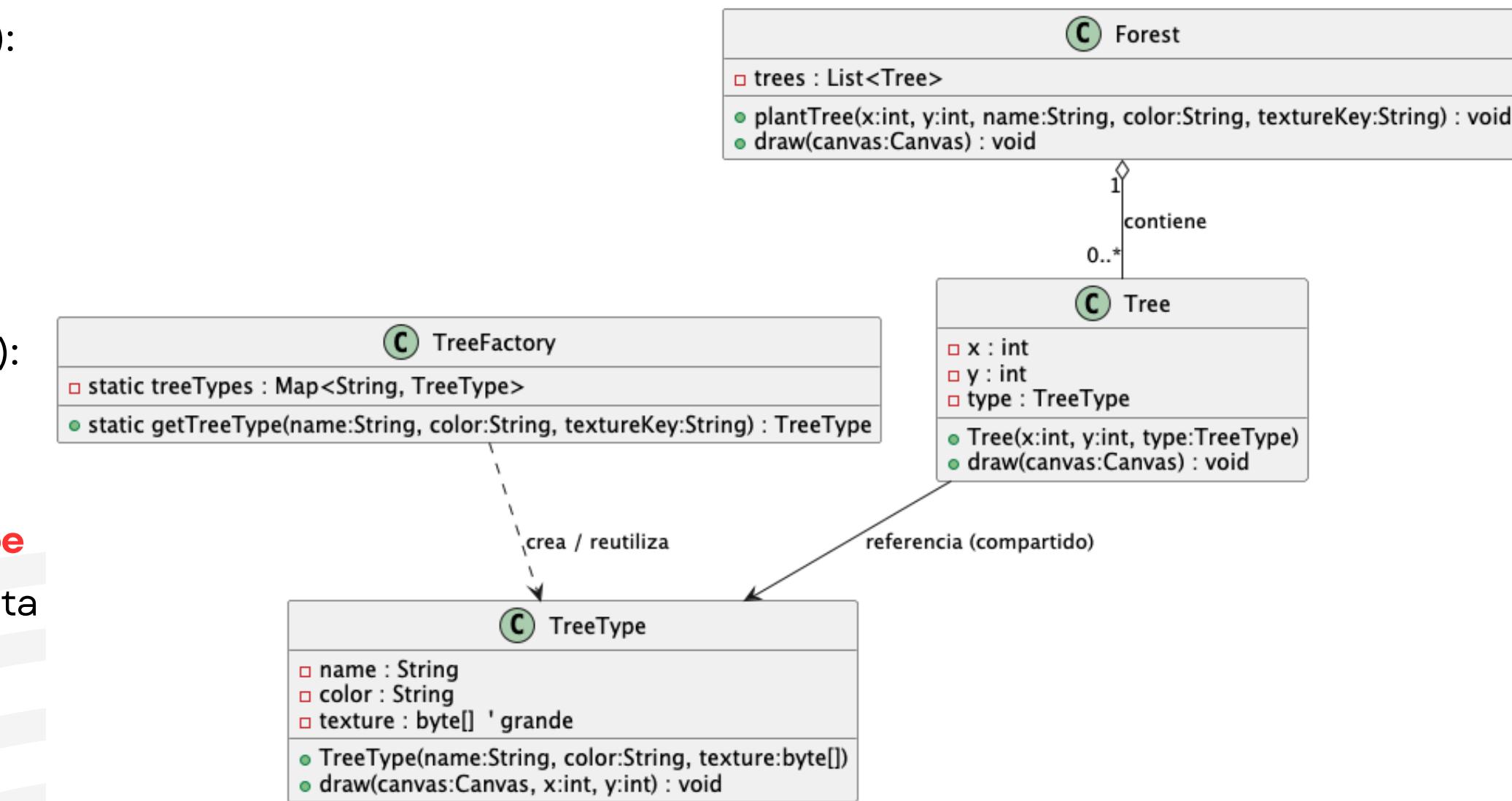
# Patrones de Diseño - Flyweight

Patrones estructurales - Peso mosca, Peso ligero, Cache



- **Estructura:**
  - **TreeType** Guarda **estado intrínseco** (compatible): **name, color, texture** (grande). Implementa comportamiento común (**draw(...)**) que recibe el **estado extrínseco** (coordenadas) como parámetros.
  - **Tree** Contiene **estado extrínseco** (no compatible): **x, y**, y referencia a un **TreeType**. Juntos (Tree + TreeType) representan el “árbol completo”.
  - **TreeFactory** Mantiene un **pool** (mapa) de **TreeType** existentes y **reutiliza** uno cuando otro árbol necesita el mismo conjunto intrínseco (mismo **name/color/textureKey**). Si no existe, lo crea.
  - **Forest** En lugar de crear **TreeType** directamente, llama a la **fábrica** en cada **plantTree(...)**, garantizando reutilización y ahorro de memoria.
- Puede haber millones de **Tree** pero pocos **TreeType**.

Flyweight Pattern: Forest / TreeType / TreeFactory



Úsalo solo si de verdad tienes **muchísimos** objetos similares y un **problema real** de memoria. Si la cardinalidad es baja o el estado compartido es pequeño, la complejidad extra no compensa.

→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

# Patrones de Diseño - Proxy

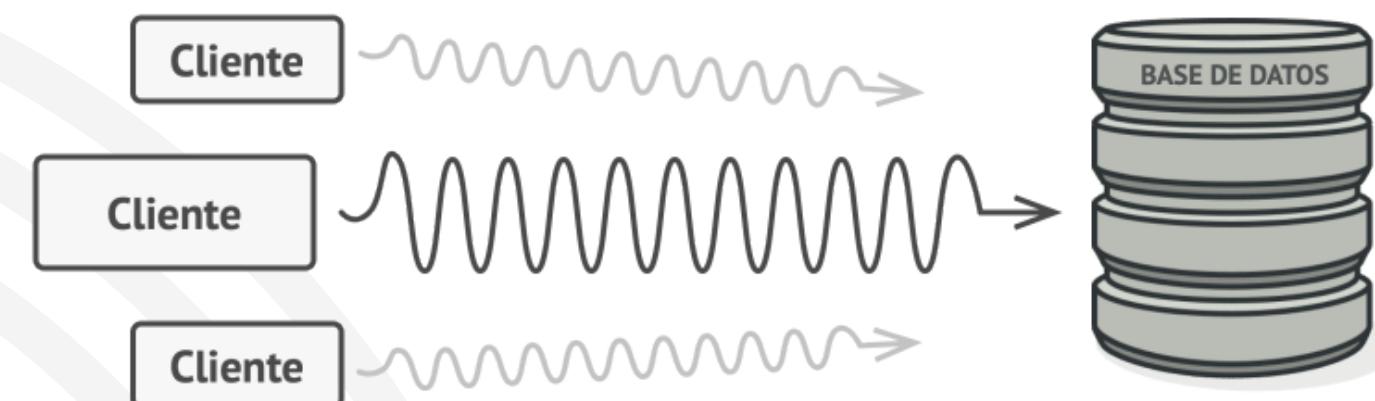
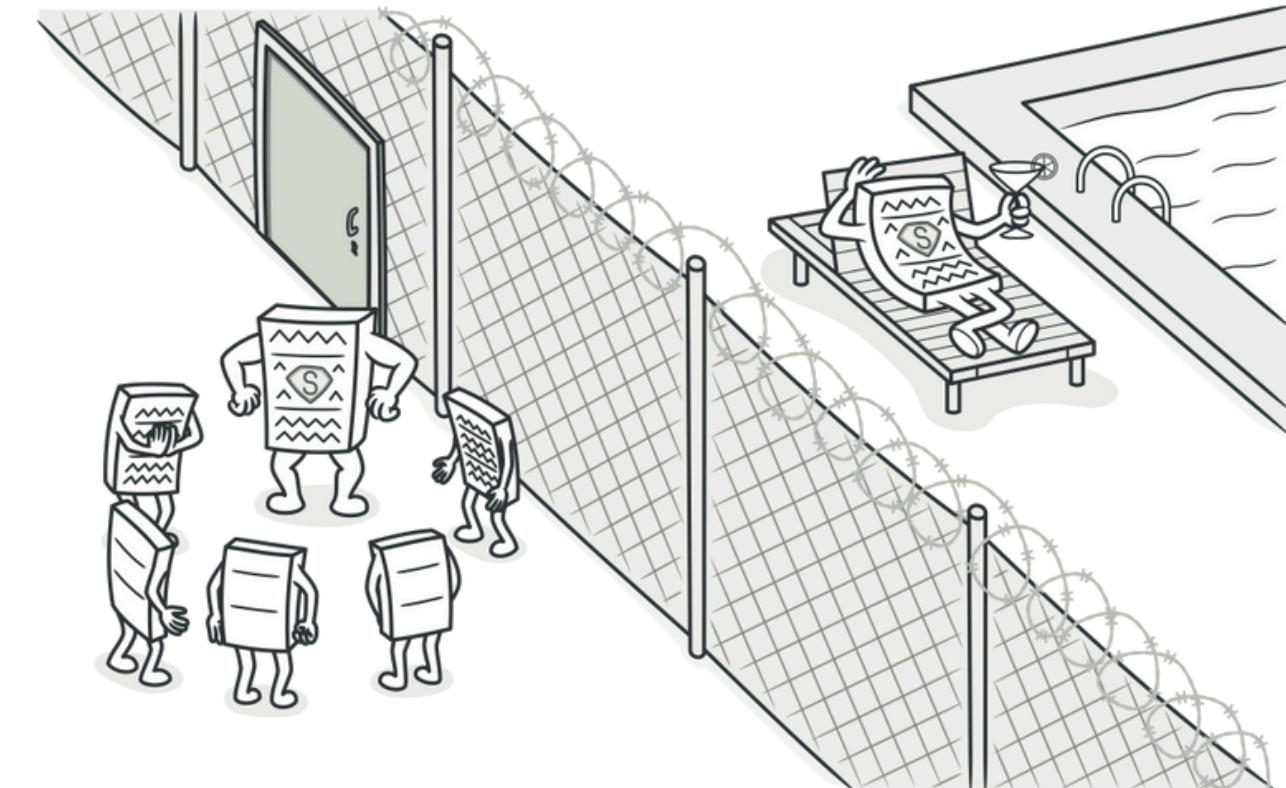
## Patrones estructurales -



**Propósito:** Es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.



**Problema:** El patrón Proxy controla el acceso a objetos costosos o externos. Permite inicialización diferida, seguridad o cacheo sin modificar la clase real, evitando duplicar código en los clientes.



# Patrones de Diseño - Proxy

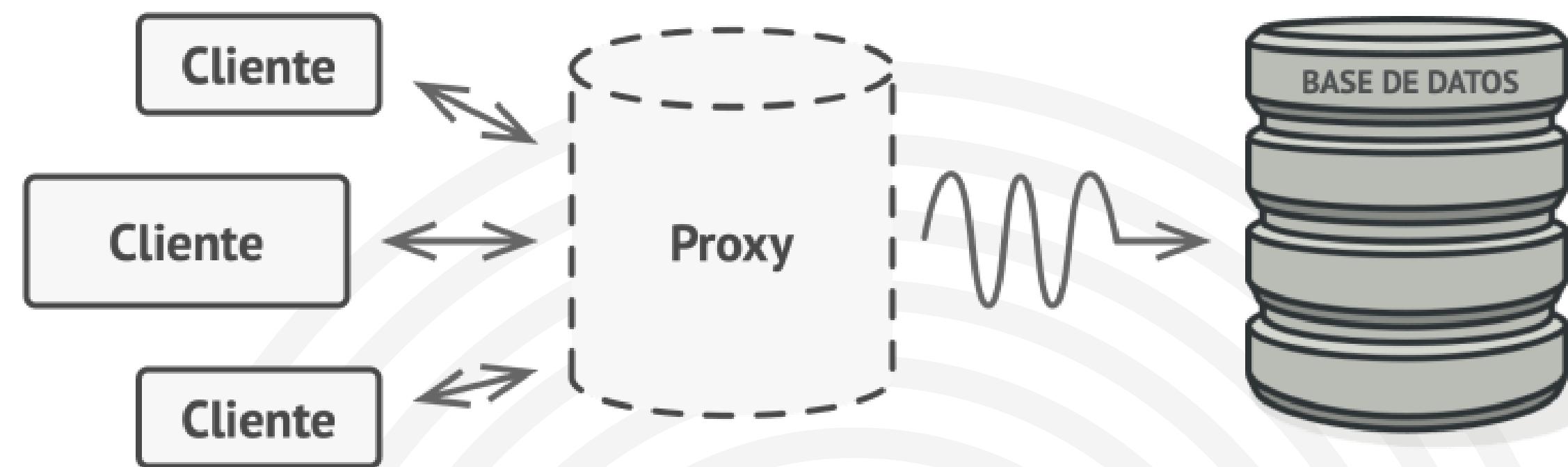
## Patrones estructurales -



**Solución:** El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

El proxy se camufla como objeto de la base de datos. Puede gestionar la inicialización diferida y el caché de resultados sin que el cliente o el objeto real de la base de datos lo sepan.

Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de

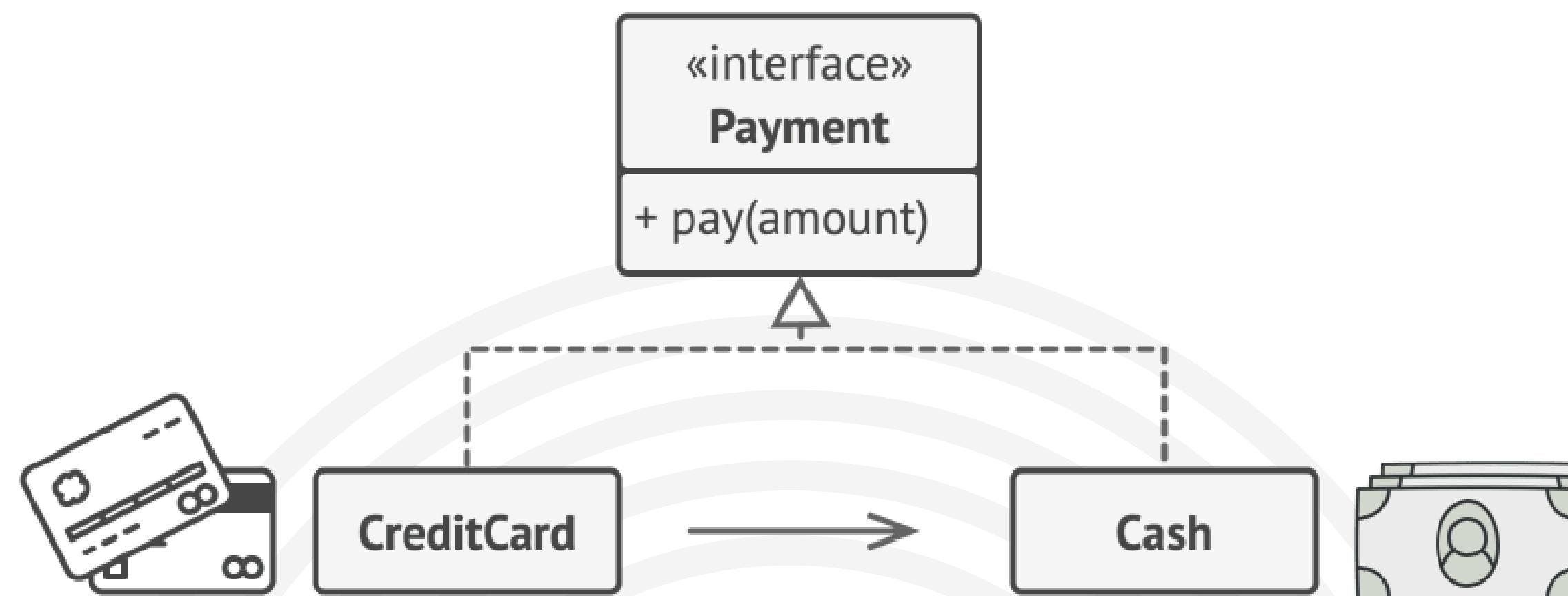


# Patrones de Diseño - Proxy

## Patrones estructurales -



**Analogía en el mundo real:** Una tarjeta de crédito es un proxy de una cuenta bancaria, que, a su vez, es un proxy de un manojo de billetes. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un pago. El consumidor se siente genial porque no necesita llevar un montón de efectivo encima. El dueño de la tienda también está contento porque los ingresos de la transacción se añaden electrónicamente a la cuenta bancaria de la tienda sin el riesgo de perder el depósito o sufrir un robo de camino al banco.



# Patrones de Diseño - Proxy

## Patrones estructurales -



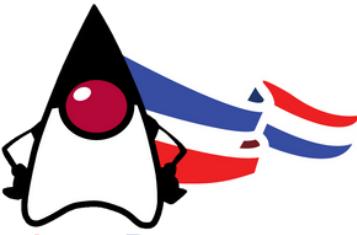
### Aplicabilidad:

- Proxy virtual: retrasa la creación de objetos pesados hasta que realmente se necesitan, ahorrando recursos.
- Proxy de protección: controla el acceso, permitiendo uso solo a clientes con credenciales válidas.
- Proxy remoto: gestiona la comunicación con objetos ubicados en servidores externos, ocultando la complejidad de red.



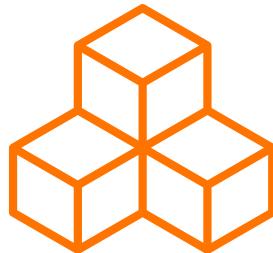
### Pros y contras:

- ✓ Proxy controla acceso, ciclo de vida y añade flexibilidad sin cambiar clientes.
- ✗ Aumenta complejidad y puede introducir retrasos en las respuestas del servicio.



# Patrones de Diseño - Proxy

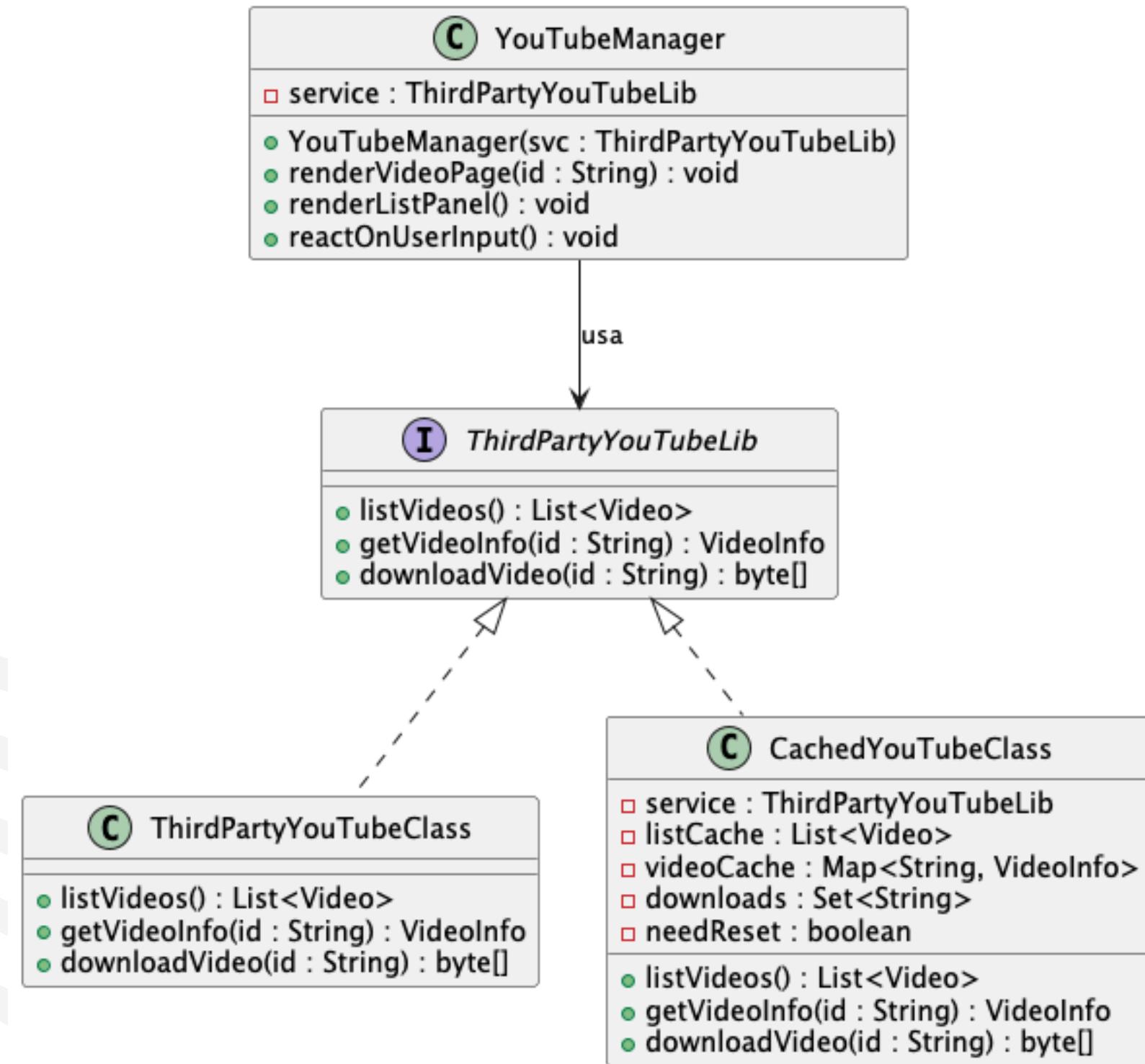
## Patrones estructurales -



- **Estructura:**
- **ThirdPartyYouTubeLib** El **Cliente** (GUI) solo conoce esta interfaz. Por eso puede recibir **un servicio real o un proxy** indistintamente.
- **ThirdPartyYouTubeClass** Implementa la integración real (llamadas remotas costosas). Es **ineficiente** si se invoca repetidamente con los mismos parámetros (descarga o consulta una y otra vez).
- **CachedYouTubeClass** implementa la **misma interfaz, delegando** al servicio real **solo** cuando es necesario. Añade **caché** (y podría añadir lazy init, control de acceso, logging, rate limit, etc.).
- **YouTubeManager** No cambia: consume la interfaz. Podemos “enchufar” el proxy sin tocar el cliente.

Beneficio clave: **mismo contrato**, comportamiento adicional transparente para el cliente (caché, lazy, etc.). Si la librería es de terceros/inmutable, el proxy te permite extender **sin modificar**.

Proxy Pattern: CachedYouTubeClass vs ThirdPartyYouTubeClass



→ Association

→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

# Muchas Gracias

Reach out to me

-  [freddy.pena@alphnology.com](mailto:freddy.pena@alphnology.com)
-  [github.com/fredpena](https://github.com/fredpena)
-  [linkedin.com/in/fred-pena/](https://linkedin.com/in/fred-pena/)
-  [x.com/fred\\_pena](https://x.com/fred_pena)

