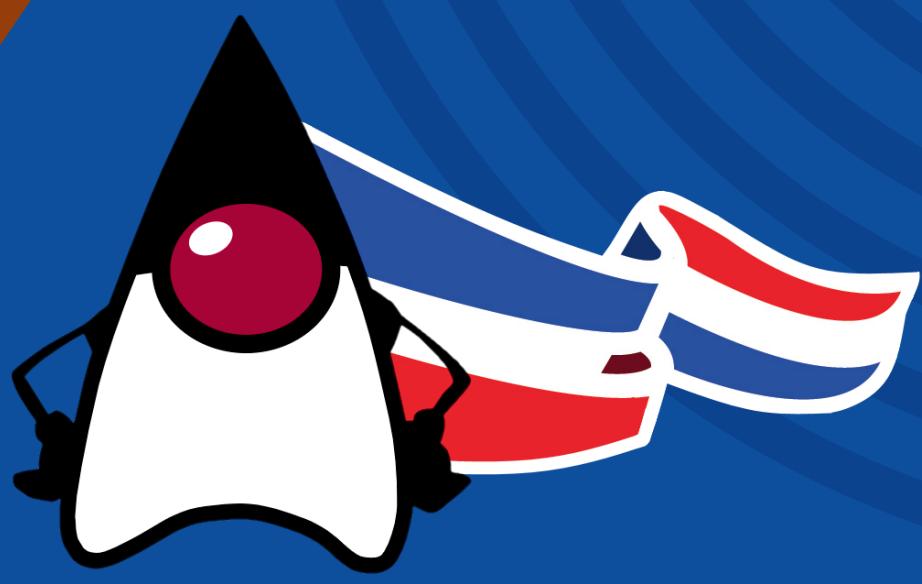




Piensa en Patrones

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



Freddy
Peña

- 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.

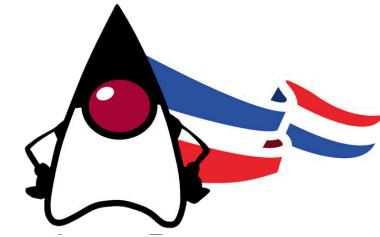
¿Qué es UML?

- **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.

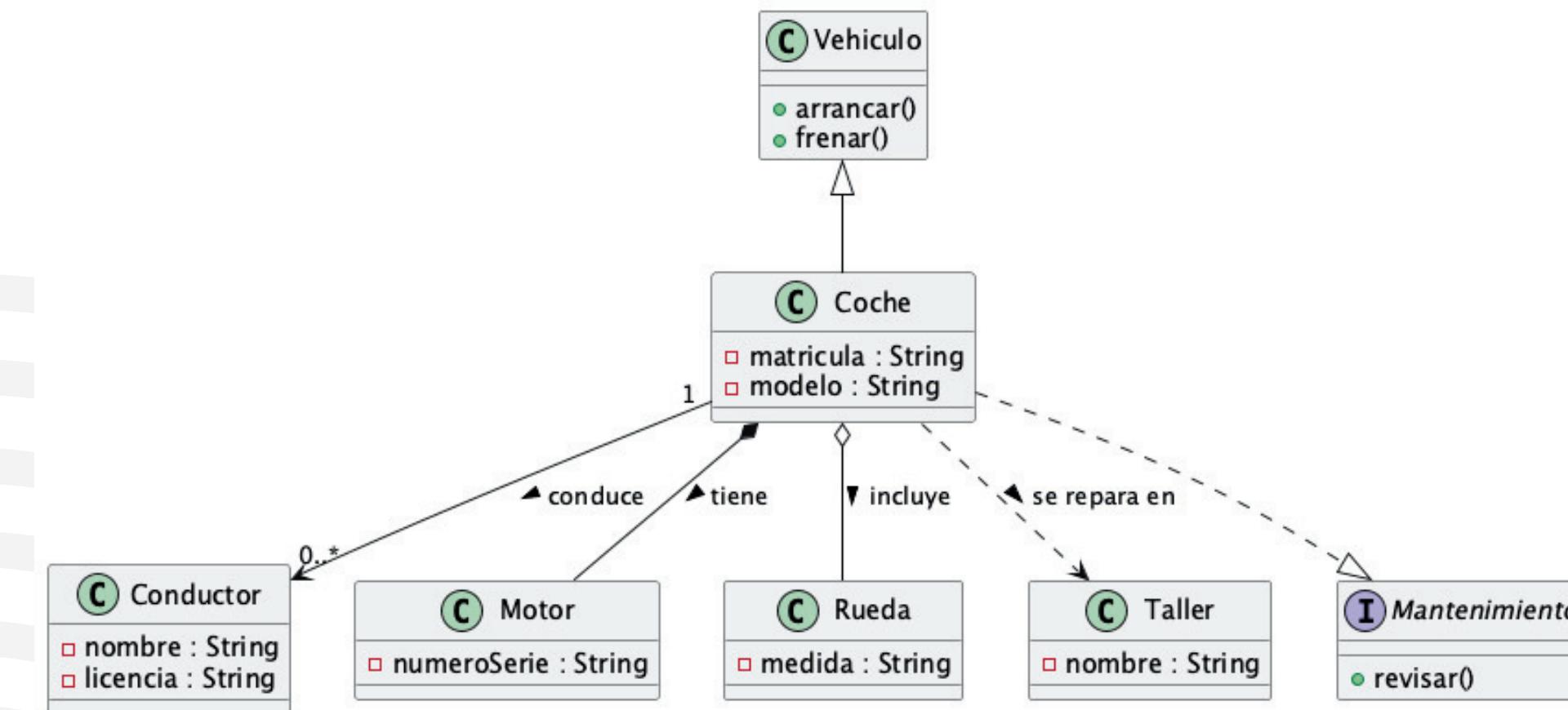


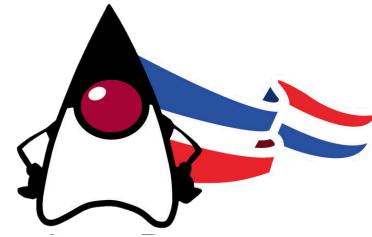
R

Diagrama de clases

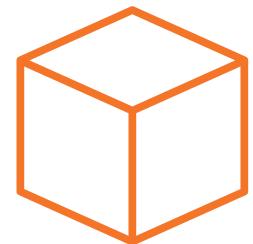


- Es uno de los **diagramas estructurales** más usados en UML. Muestra clases, atributos, operaciones (métodos) y relaciones estáticas entre clases.
- Permite entender la estructura del sistema: qué clases existen, cómo se relacionan, qué responsabilidades tienen.
- Una clase se representa típicamente 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.



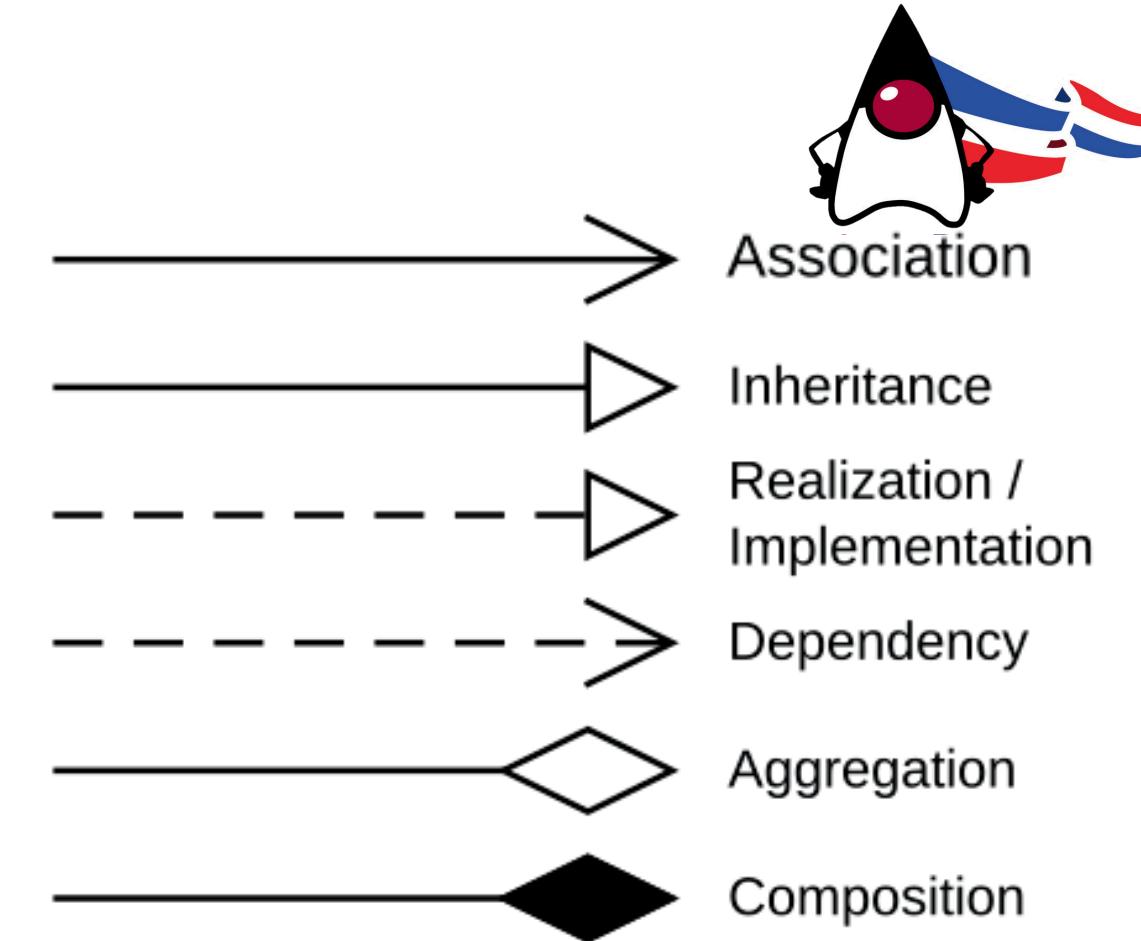
Interfaces: similares a clases, pero sólo con firmas de métodos que luego concretan otras clases.



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

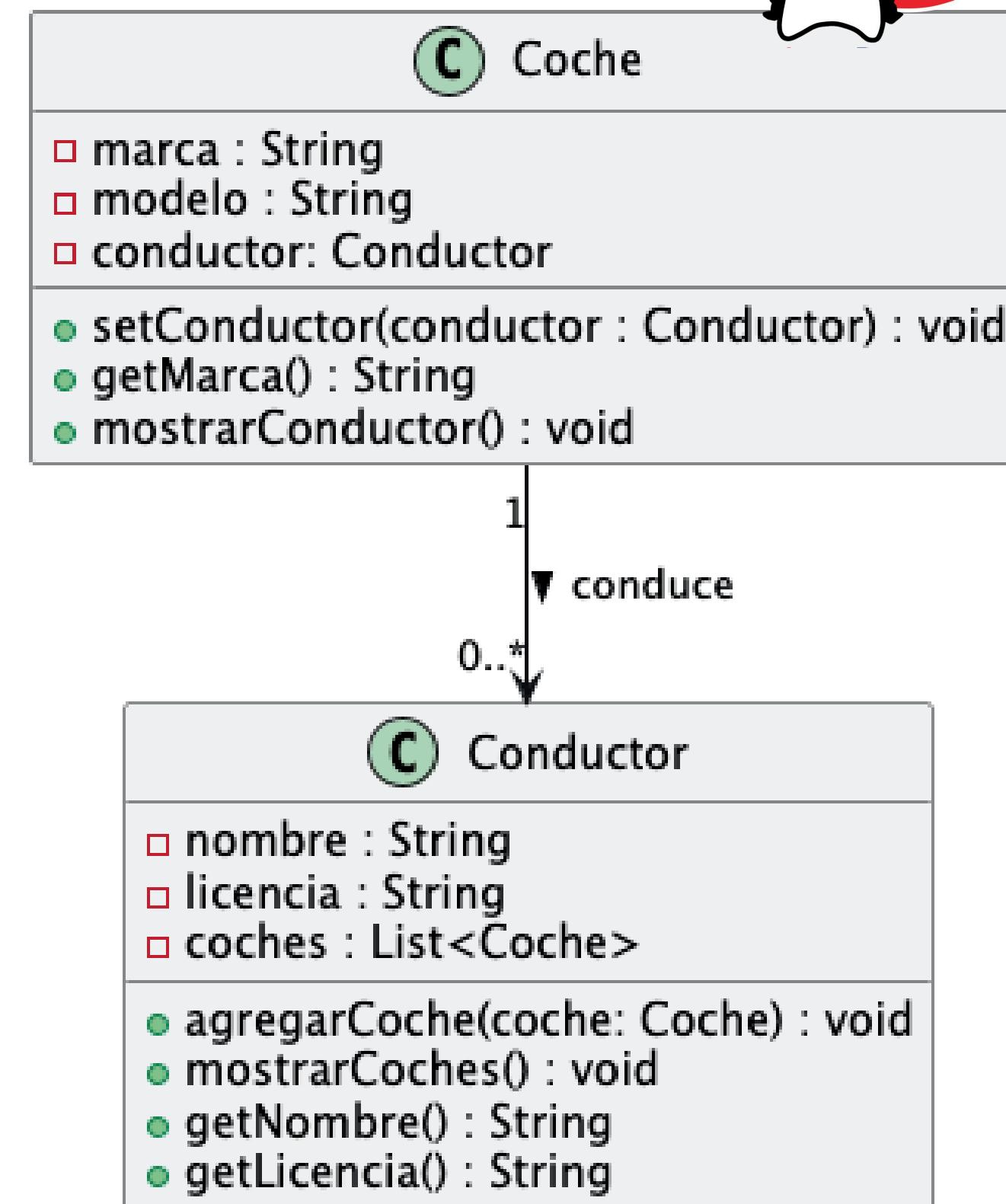
0	No hay casos (raros)
0..1	Ninguna instancia, o una sola instancia
1	Exactamente un caso
1..1	Exactamente un caso
0..*	Cero o más instancias
*	Cero o más instancias
1..*	Una o más instancias



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

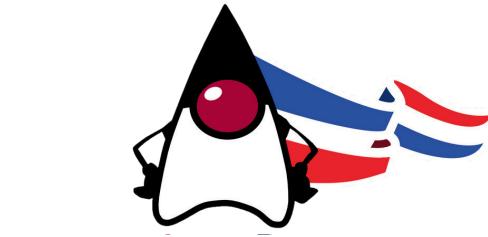
→ Aggregation

→ Composition

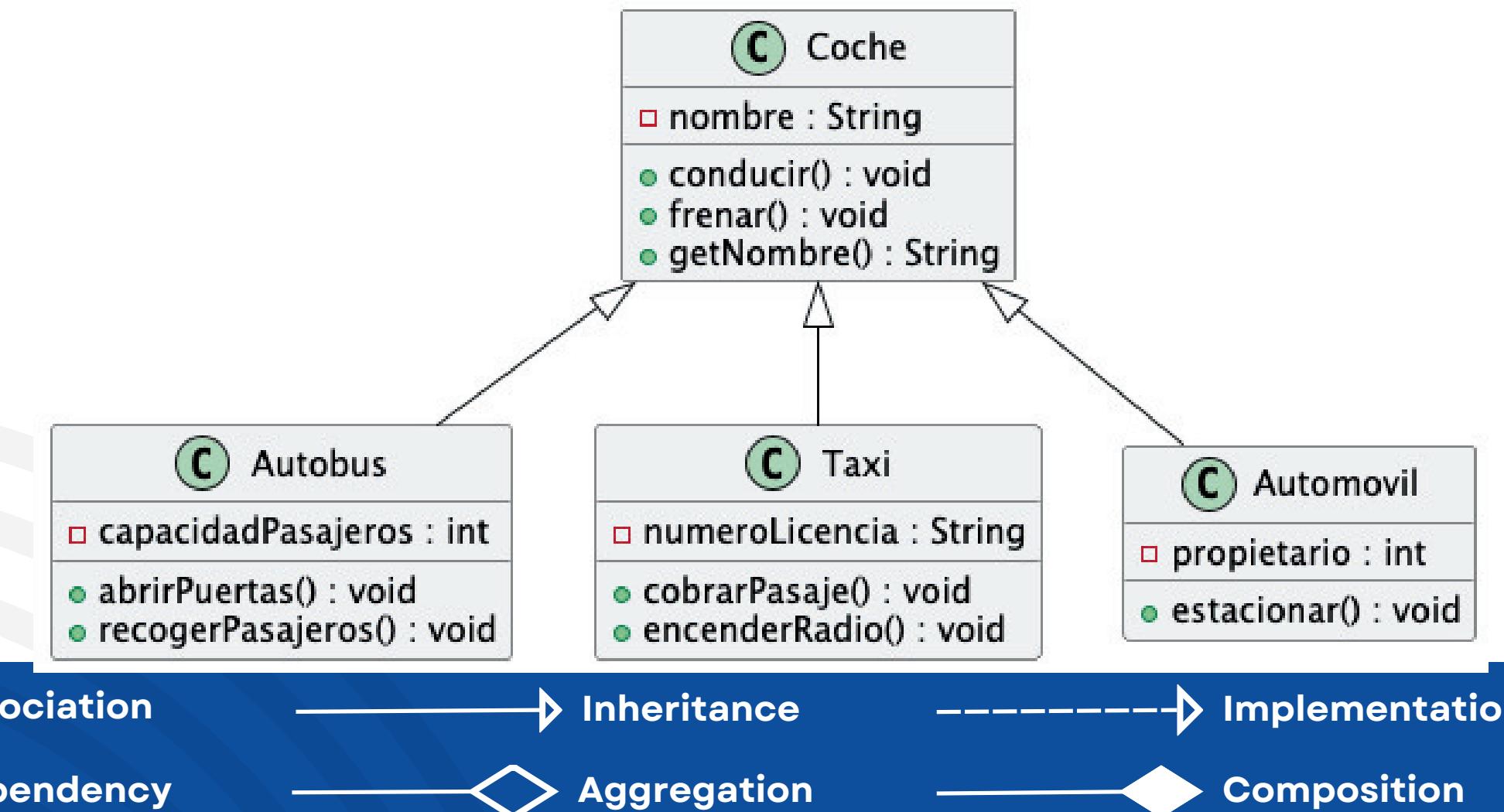


Tipos de relaciones entre clases

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.

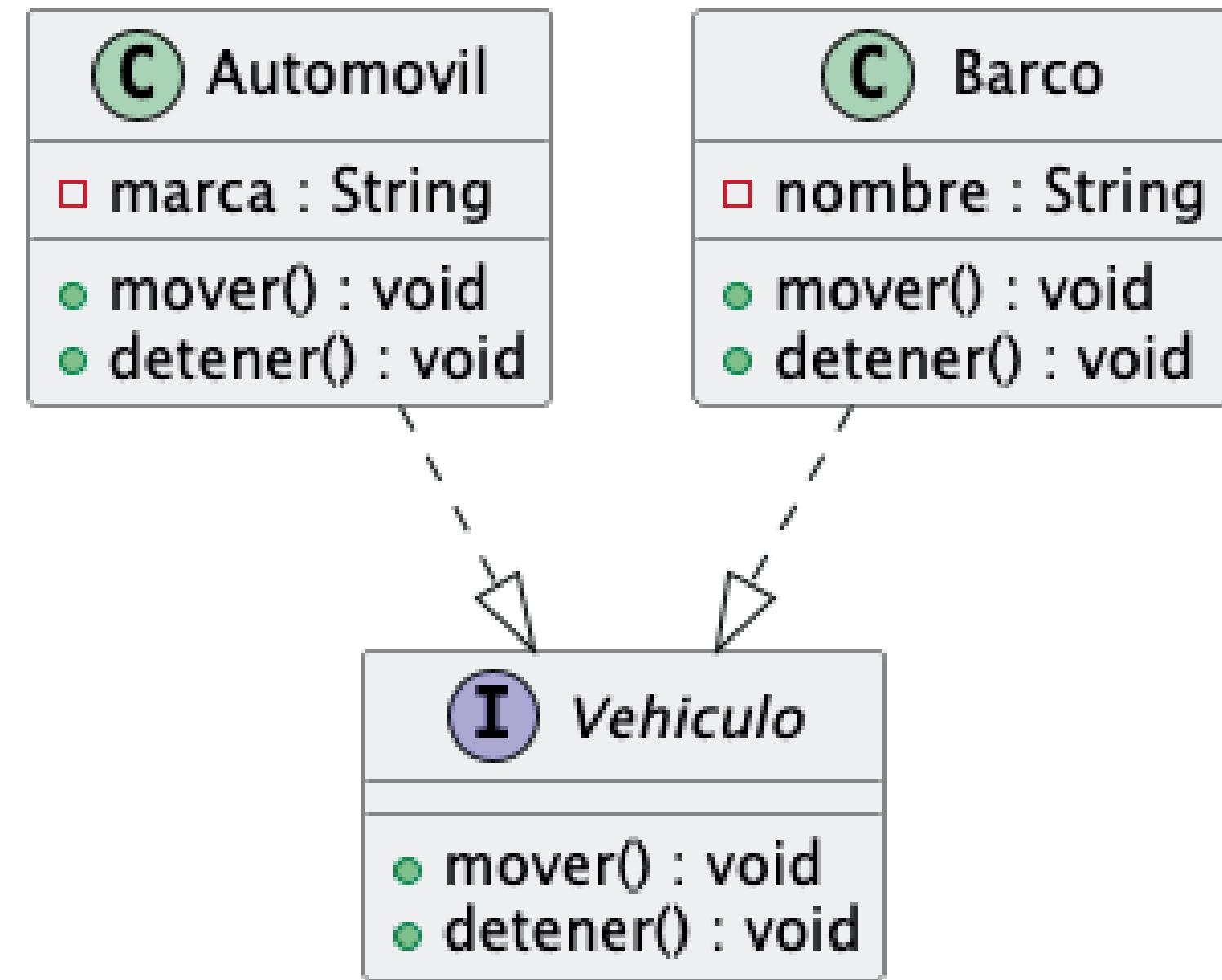




Tipos de relaciones entre clases

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

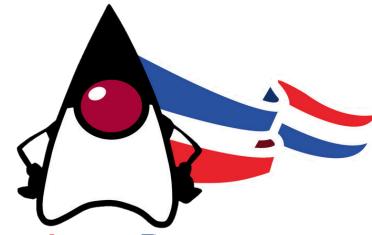
→ Dependency

→ Inheritance

→ Aggregation

→ Implementation

→ Composition

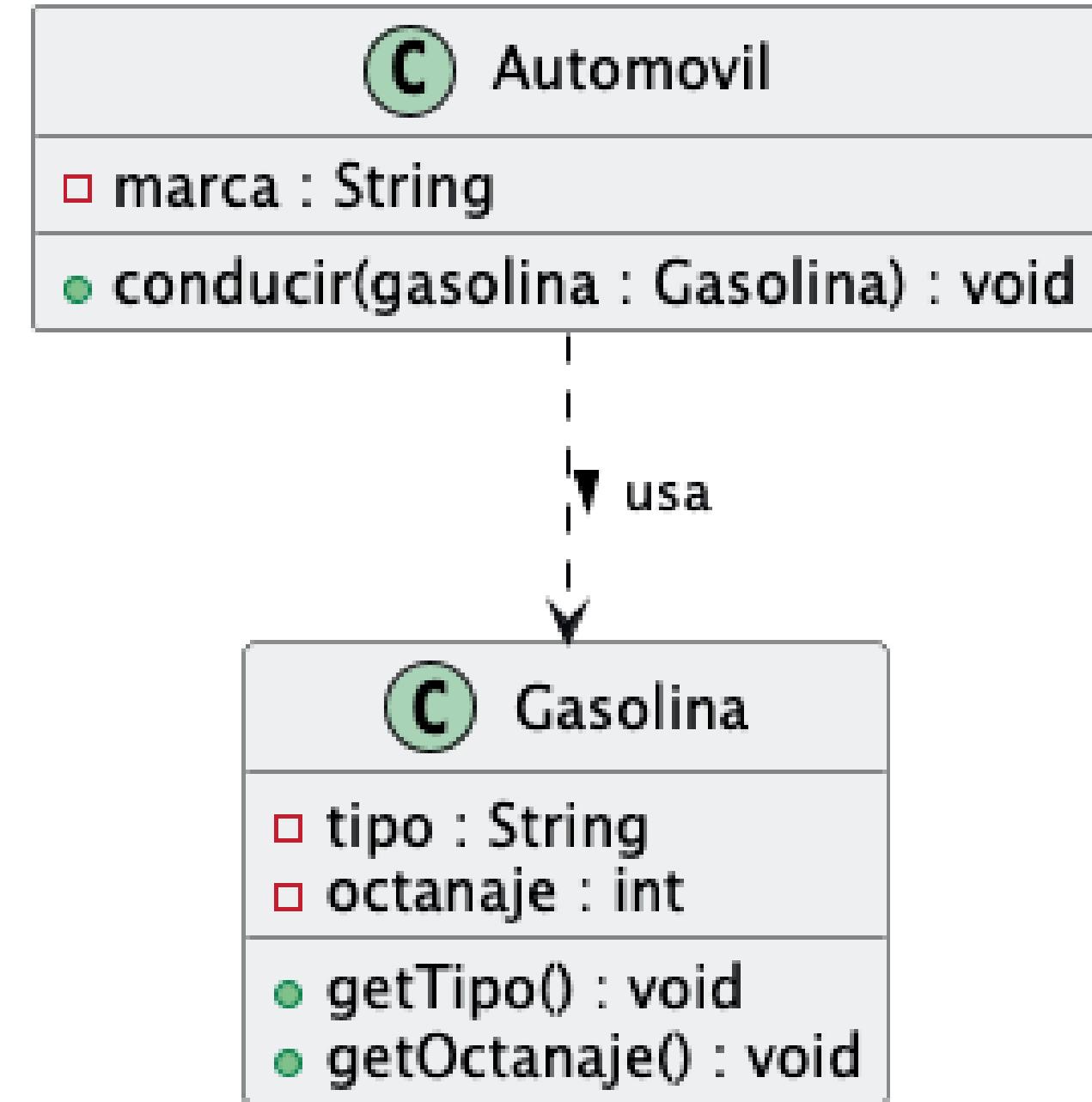




Tipos de relaciones entre clases

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

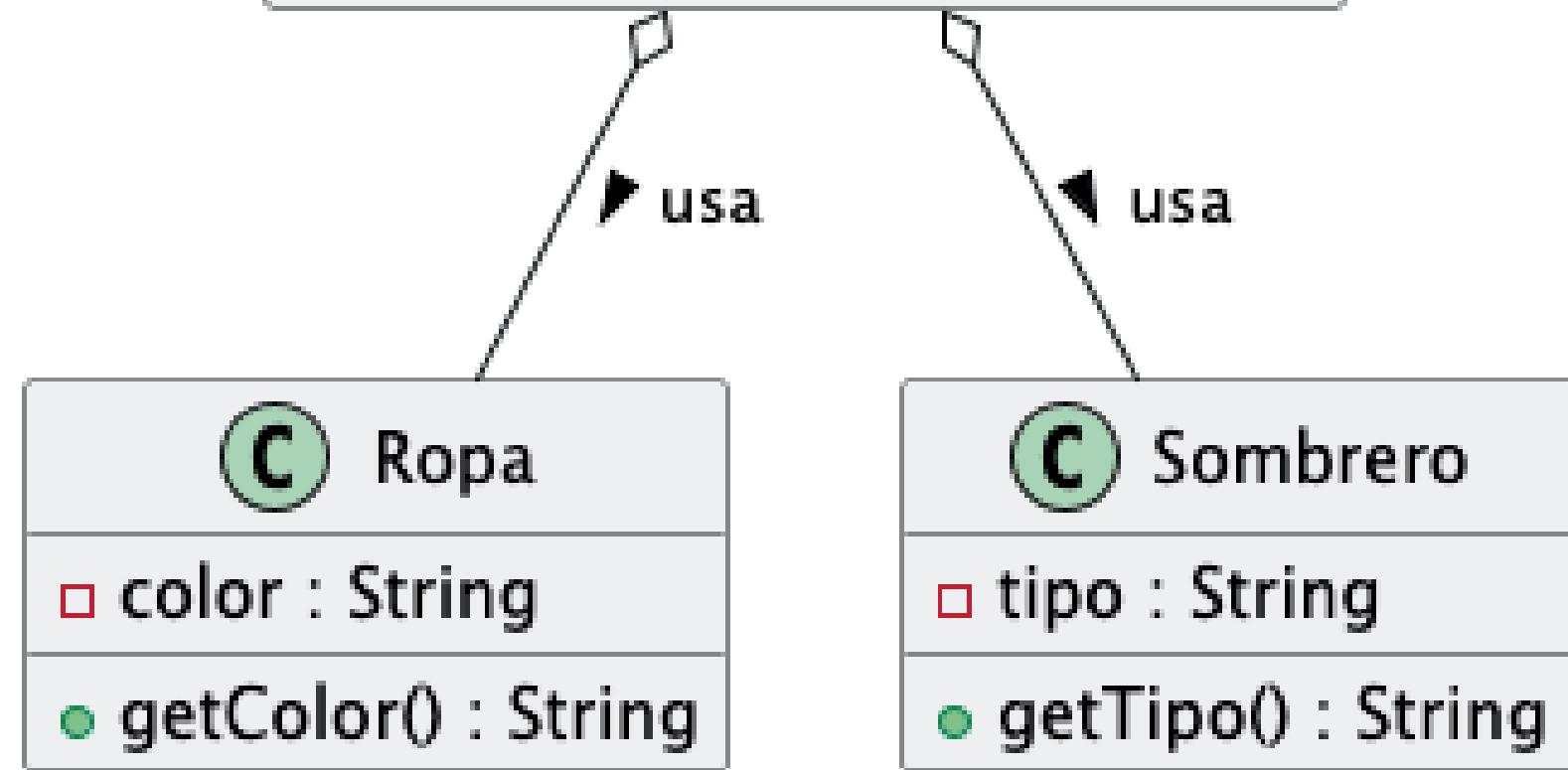
→ Composition



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

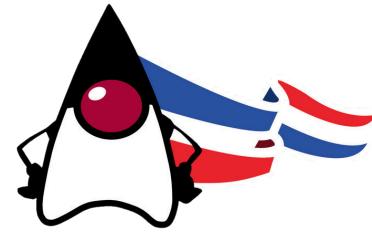
→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition

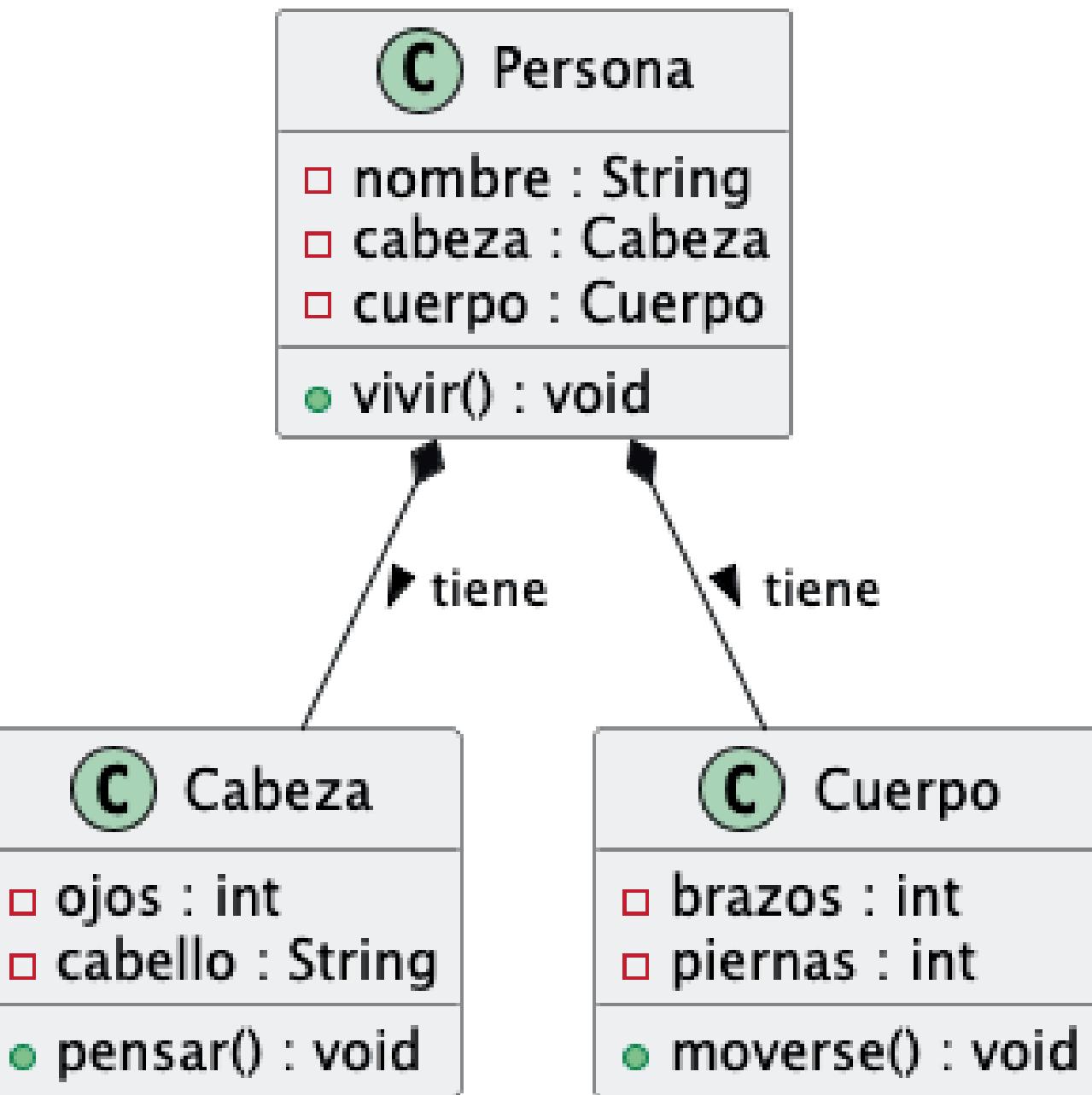




Tipos de relaciones entre clases

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

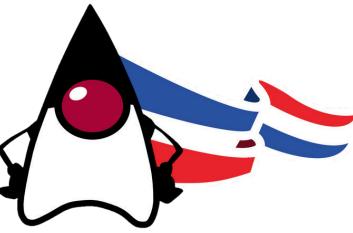
→ Inheritance

→ 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.

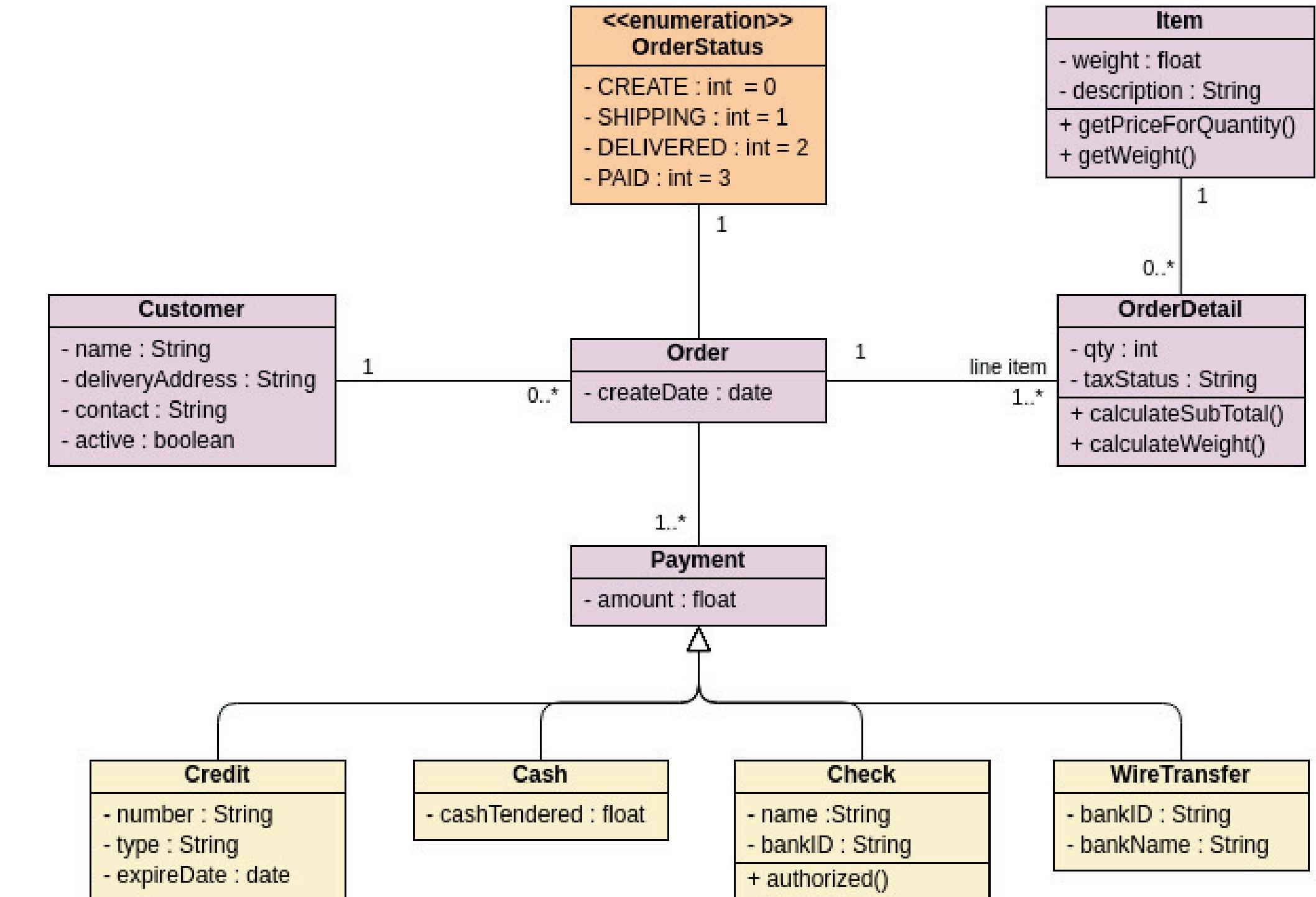


Ventajas de usar diagramas de clases bien diseñadas

- 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

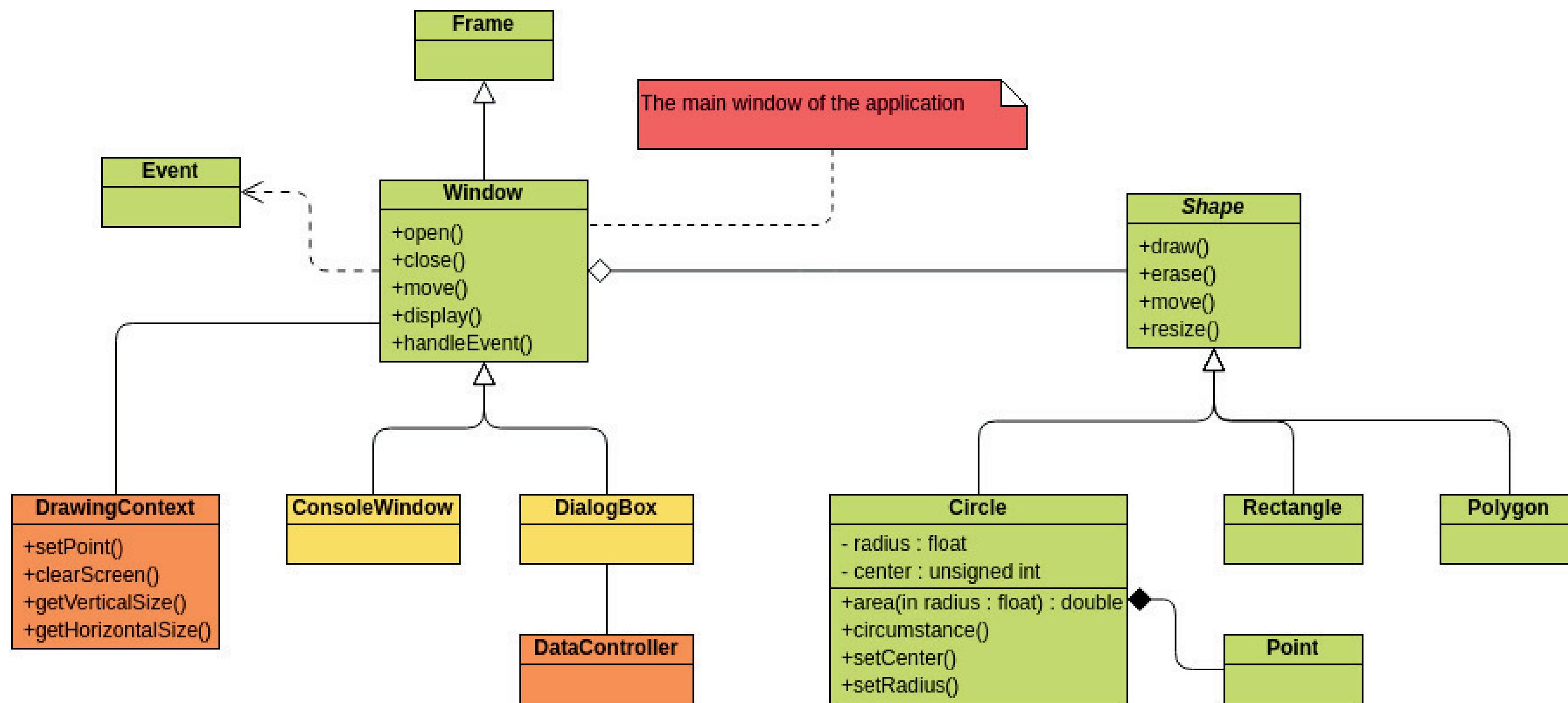
→ Dependency

→ 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**

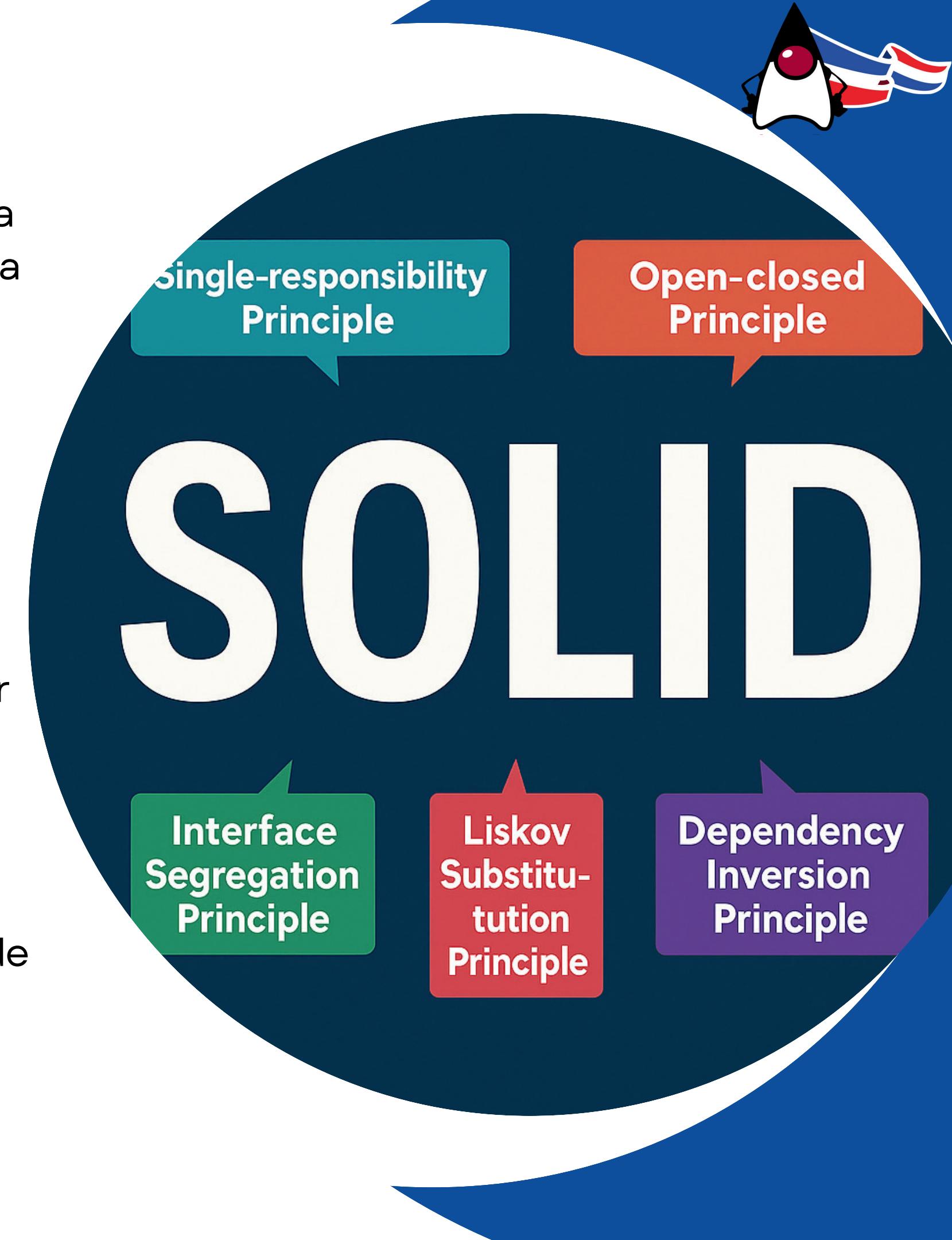


Principios SOLID

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.

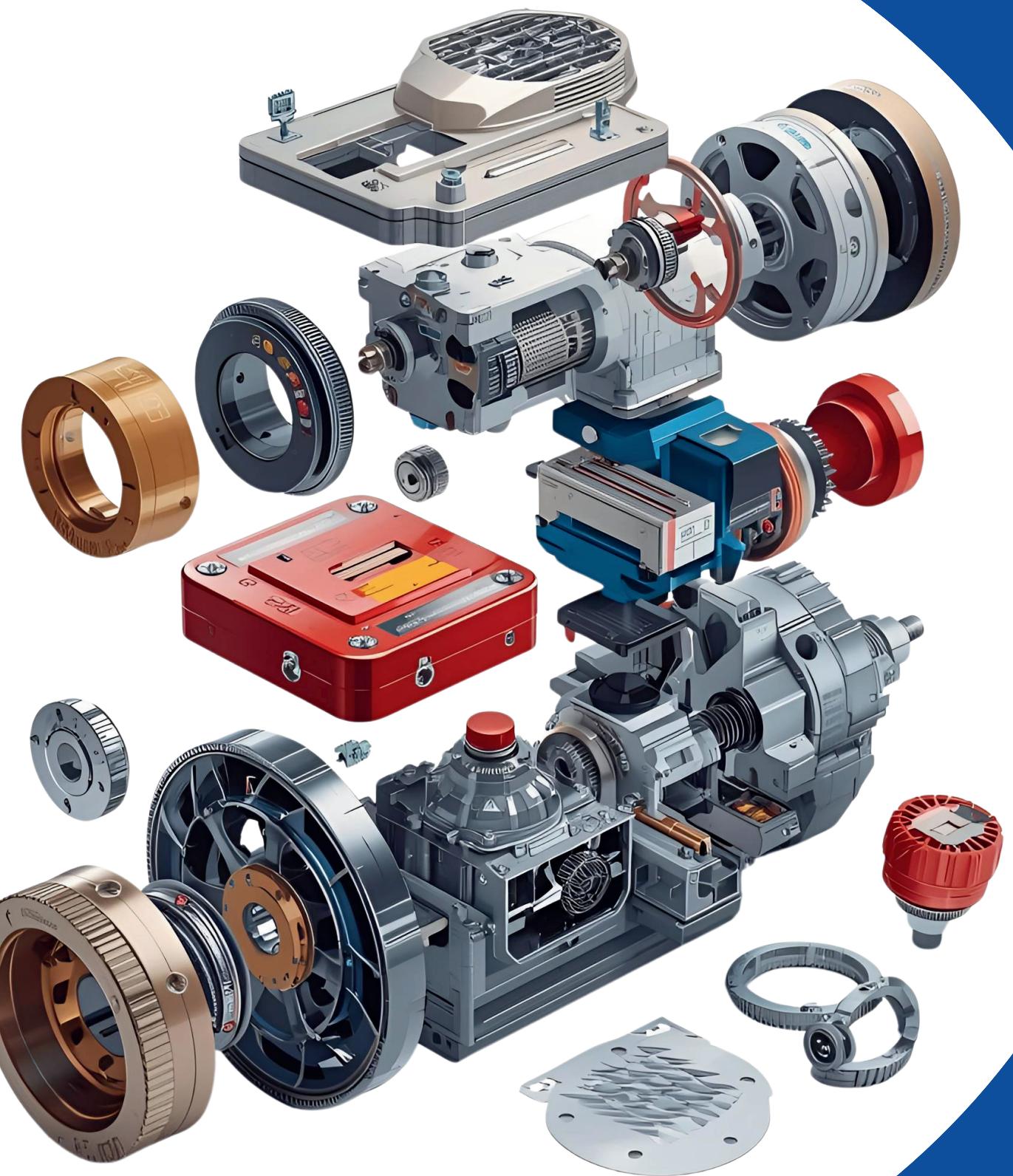




Patrones de Diseño

¿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

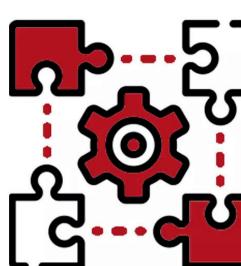
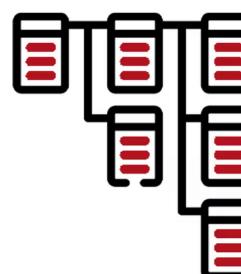
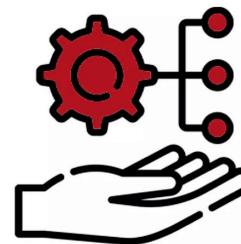
Clasificación de los patrones

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.

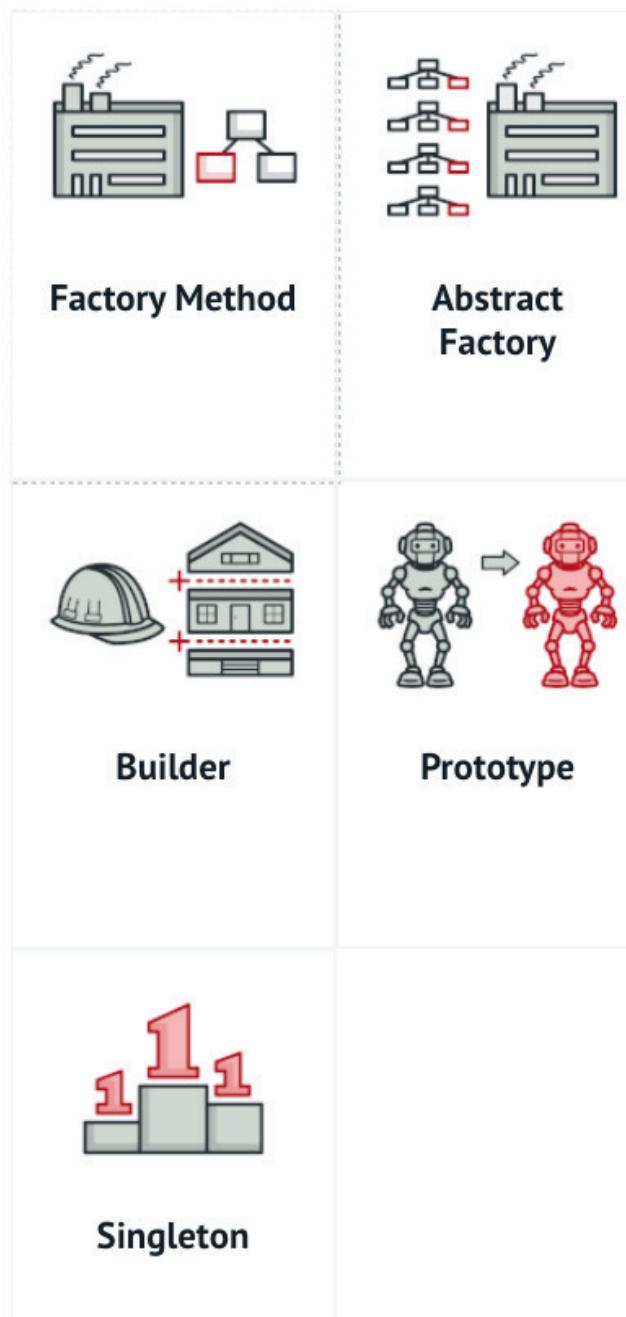
Estructurales: Explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

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

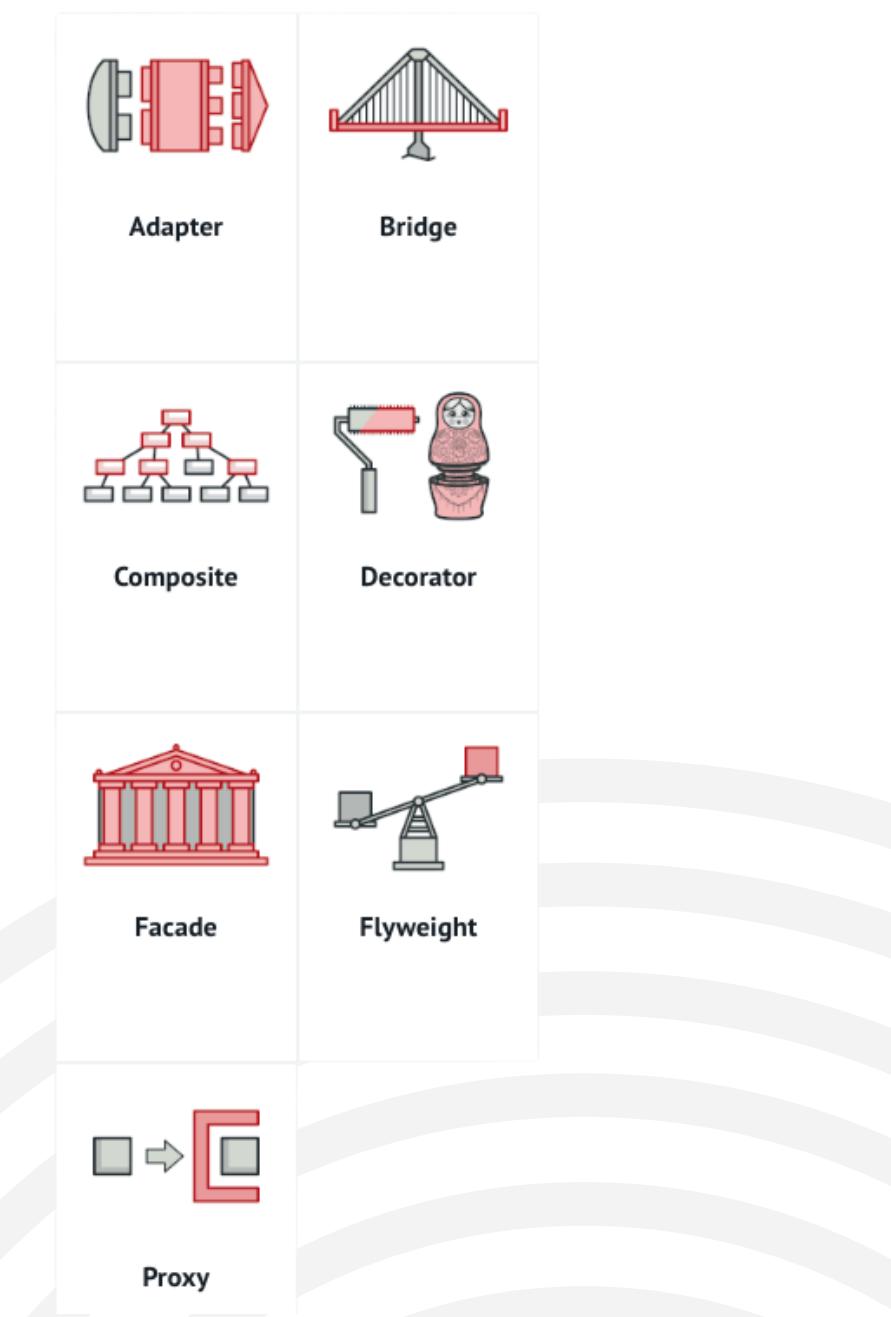
Patrones de Diseño

Clasificación de los patrones

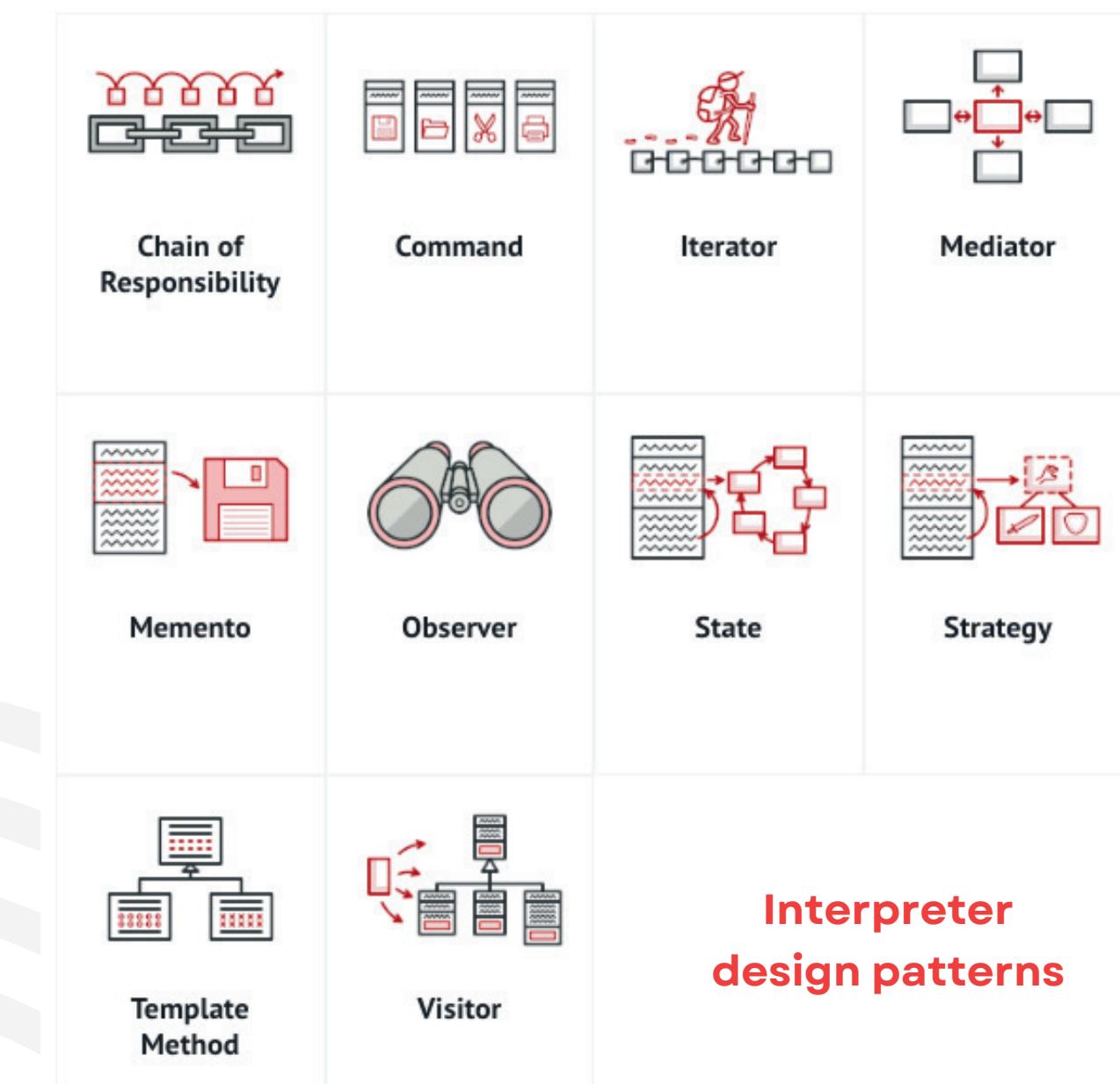
Patrones creacionales



Patrones estructurales



Patrones de comportamiento

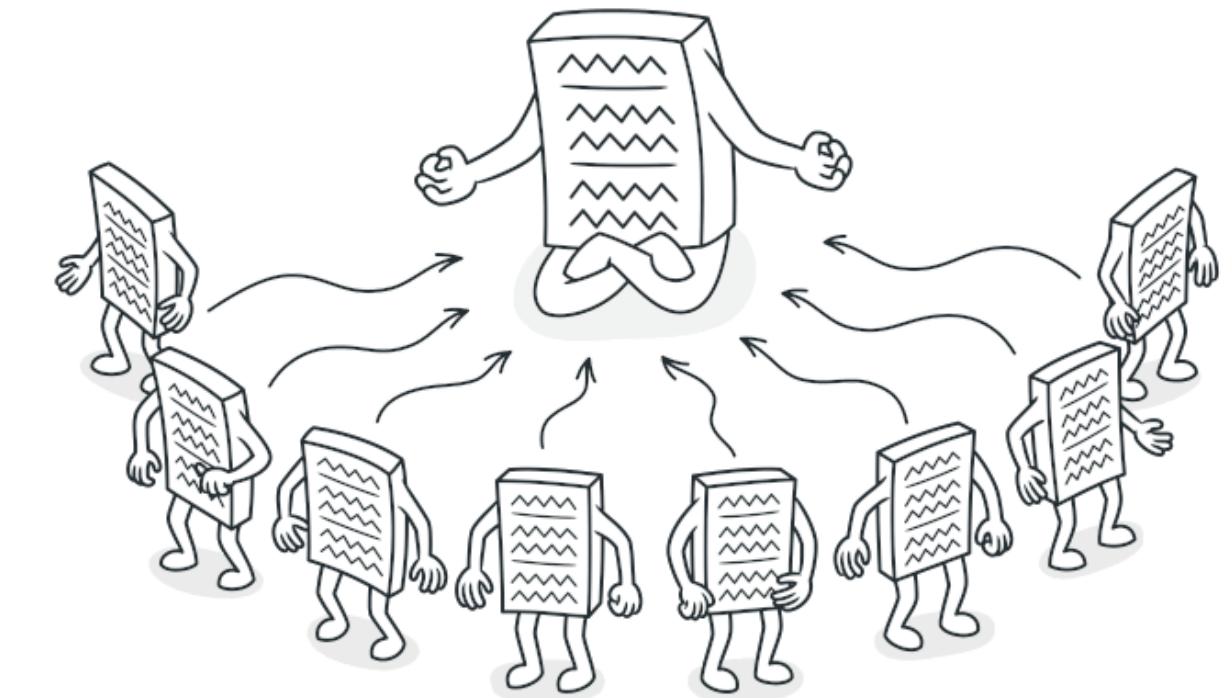


Patrones de Diseño - Singleton

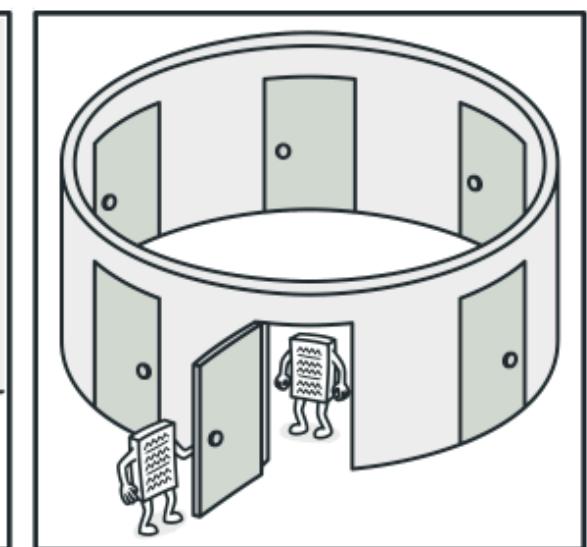
Patrones creacionales - Instancia única



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.

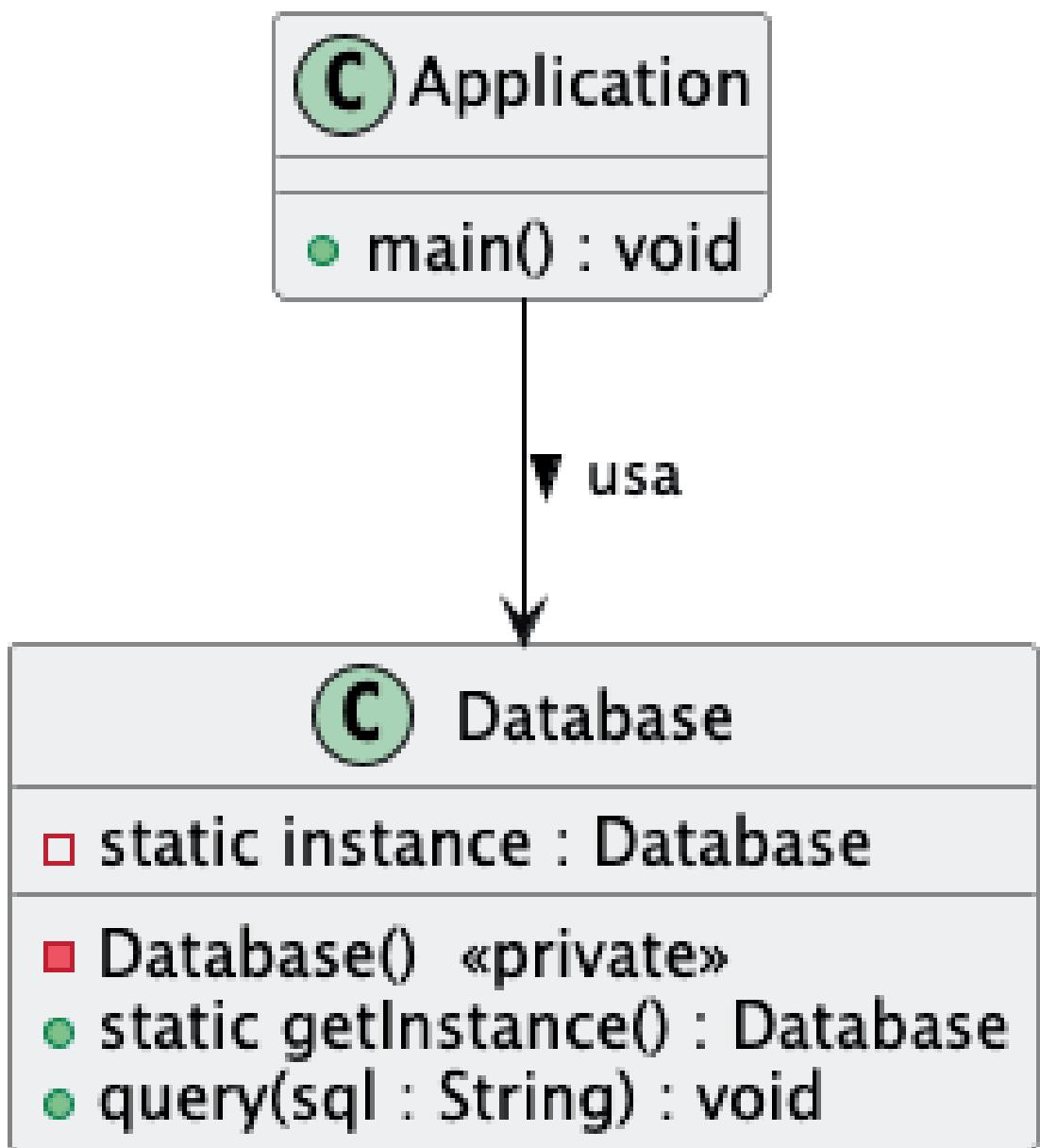
Patrones de Diseño - Singleton

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

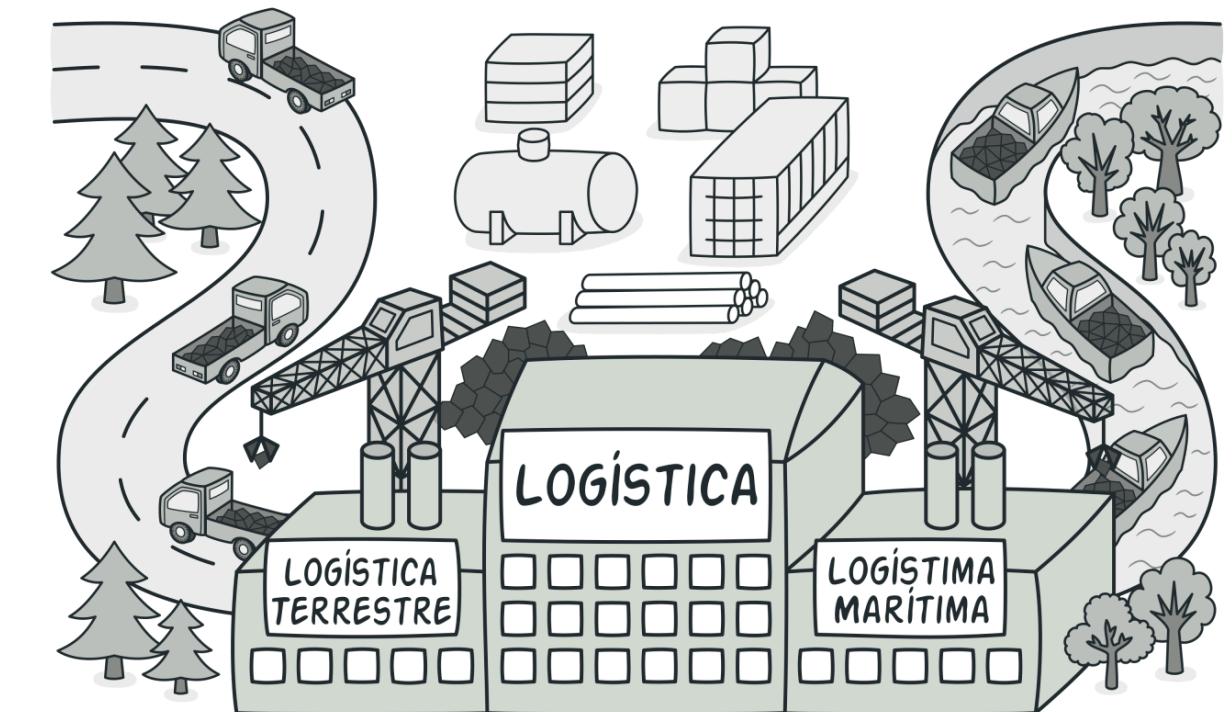
→ Composition

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.

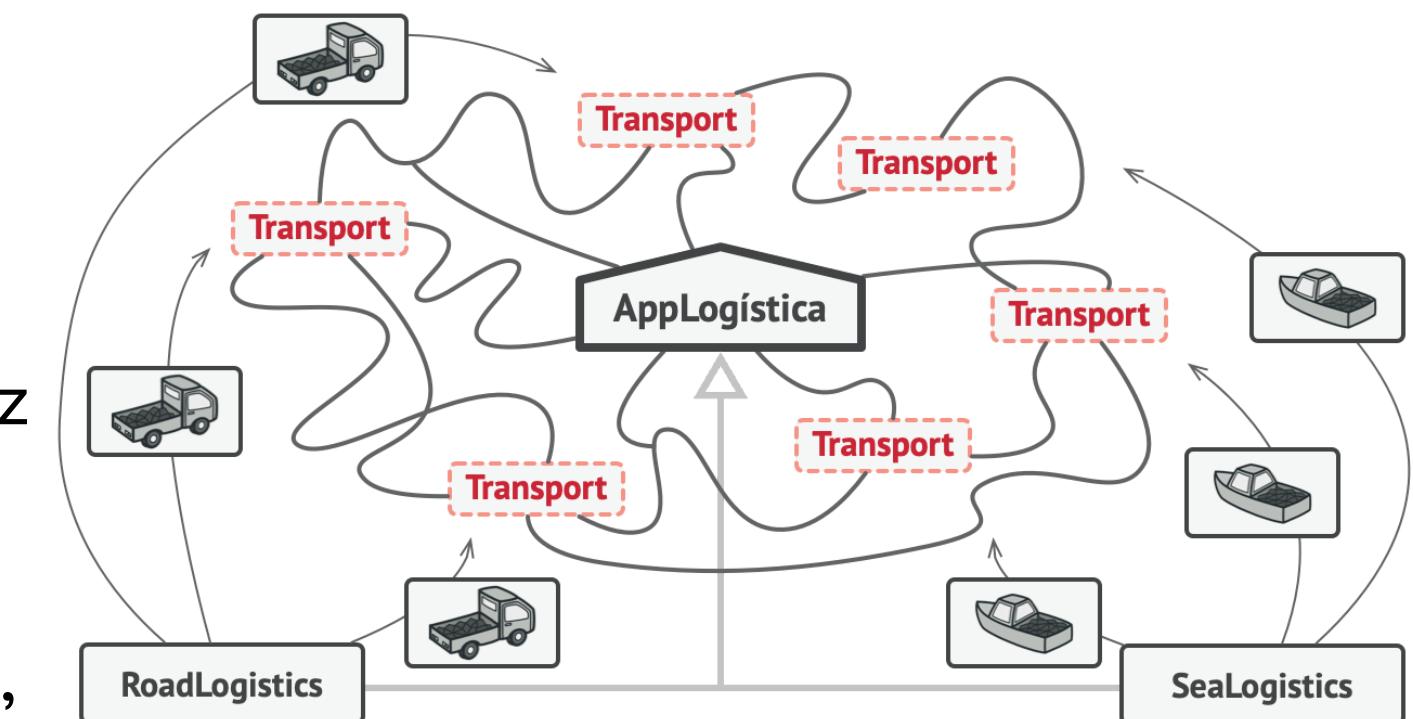


Patrones de Diseño - Factory Method

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



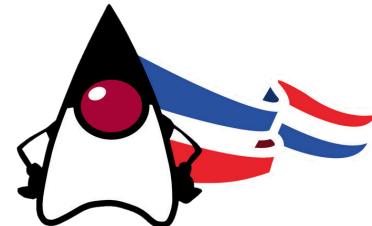
Aplicabilidad:

- 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.



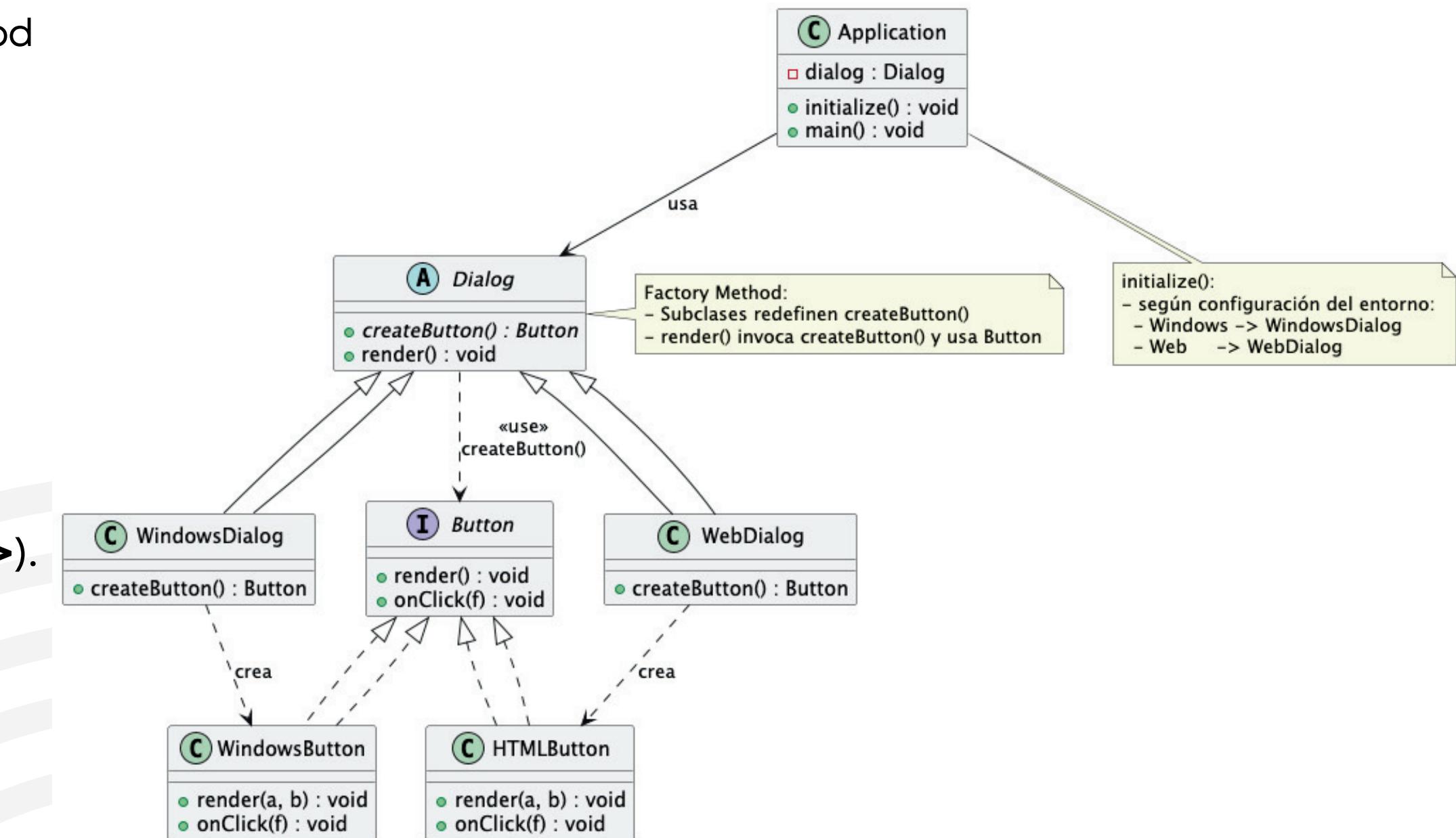
Patrones de Diseño - Factory Method

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.

Factory Method: Diálogo y Botones multiplataforma



→ Association

→ Dependency

→ Inheritance

→ Aggregation

→ Implementation

→ Composition

Patrones de Diseño - Abstract Factory

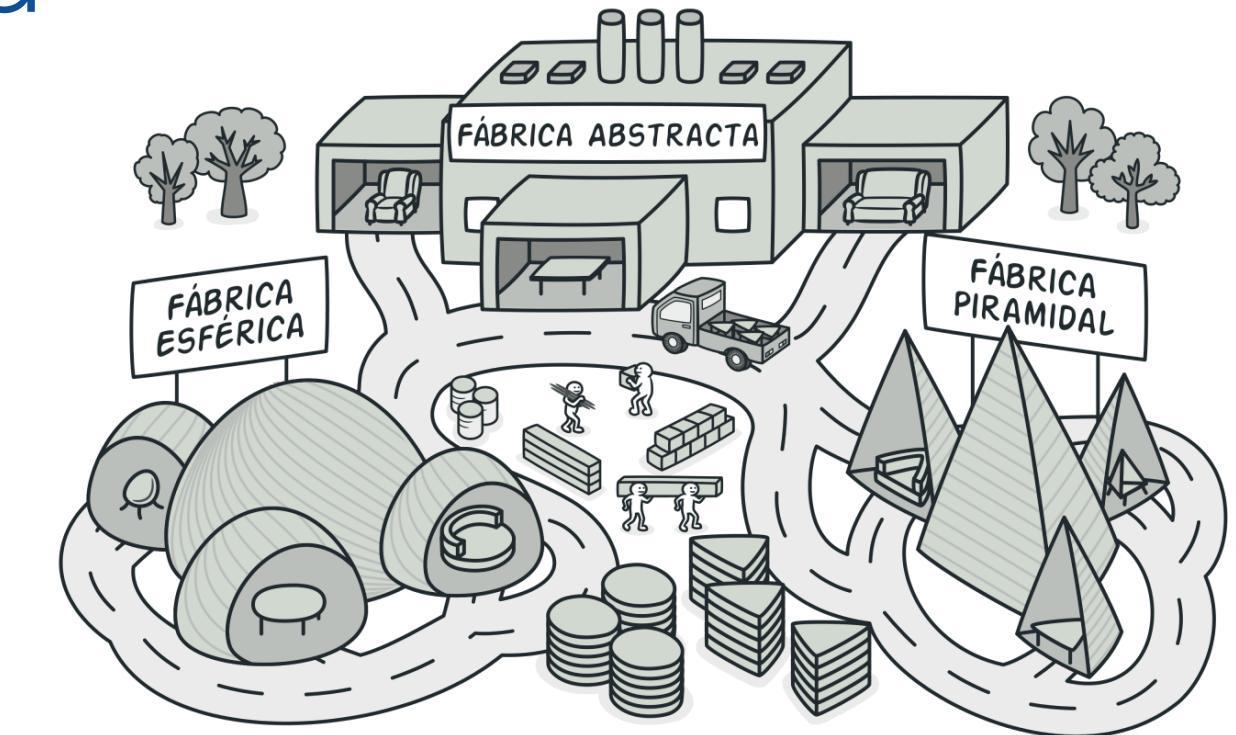
Patrones creacionales - Fábrica abstracta

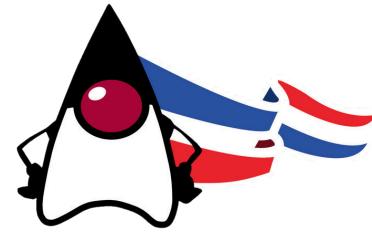


Propósito: Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



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.



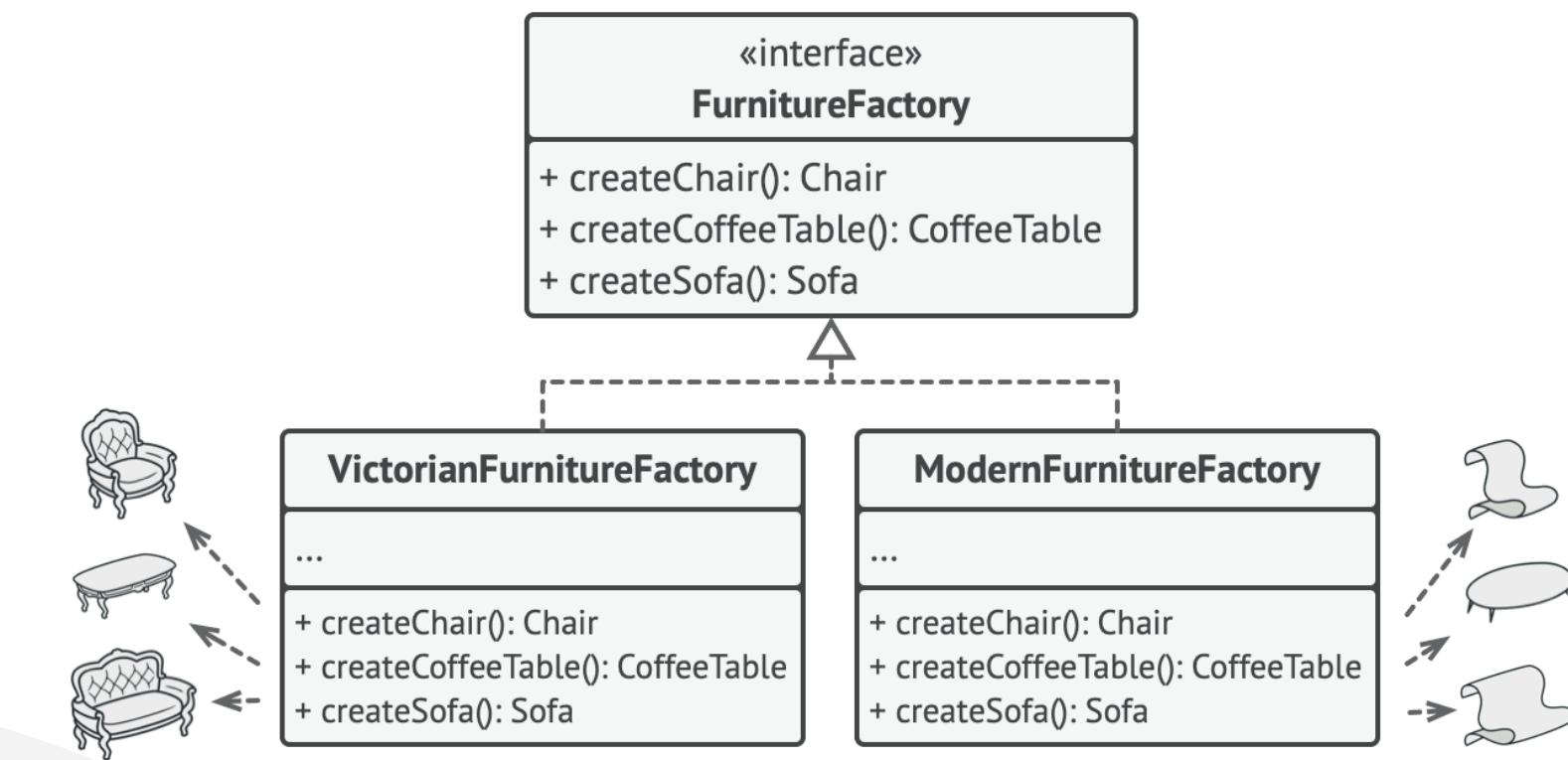


Patrones de Diseño - Abstract Factory

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



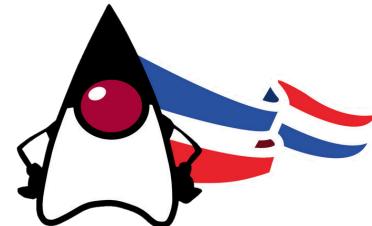
Aplicabilidad:

- 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.



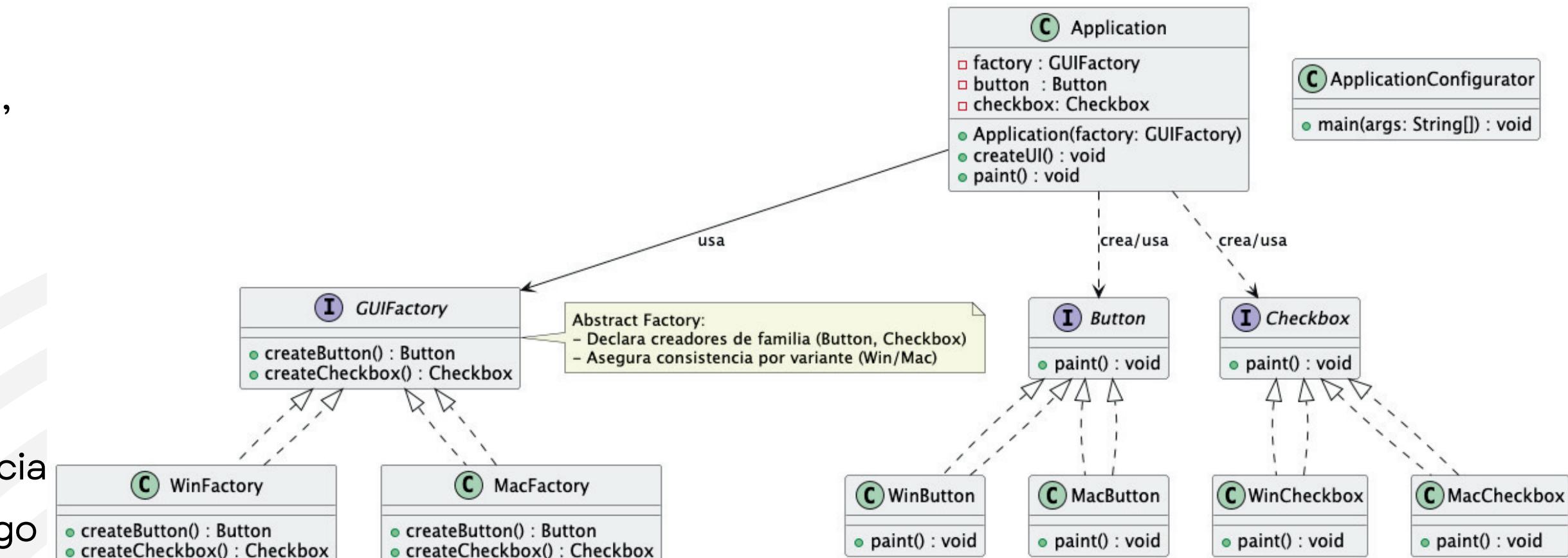
Patrones de Diseño - Abstract Factory

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

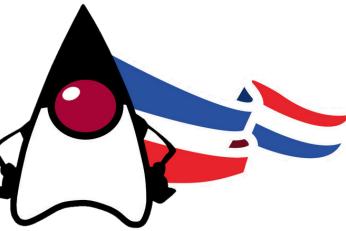
→ Inheritance

→ Implementation

→ Dependency

→ Aggregation

→ Composition



Patrones de Diseño - Builder

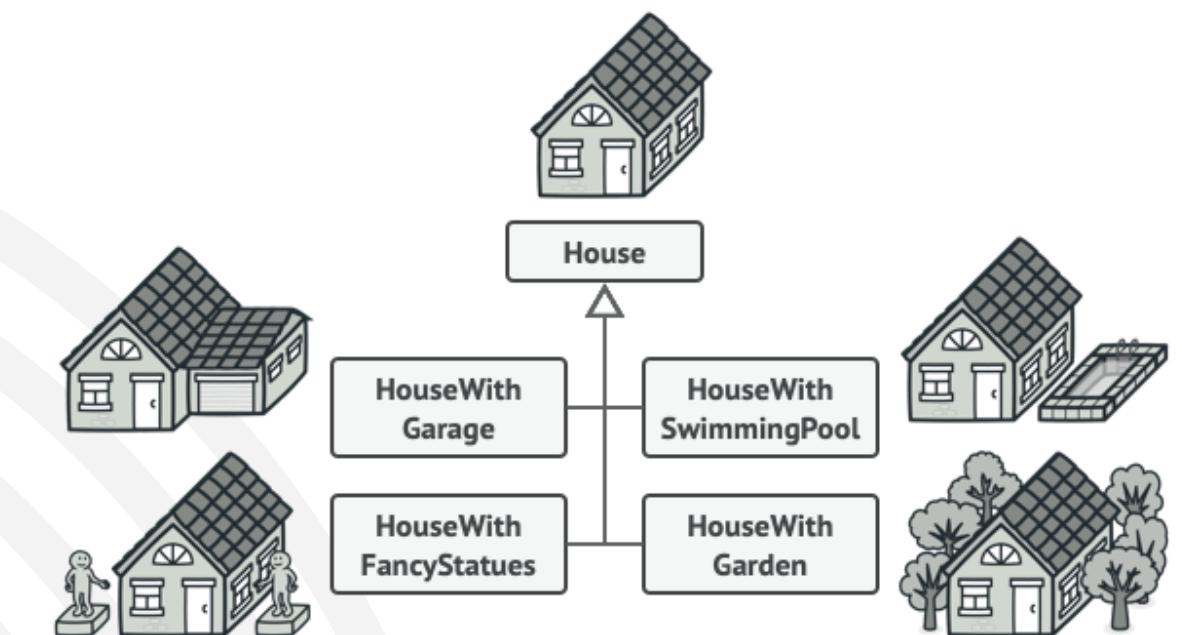
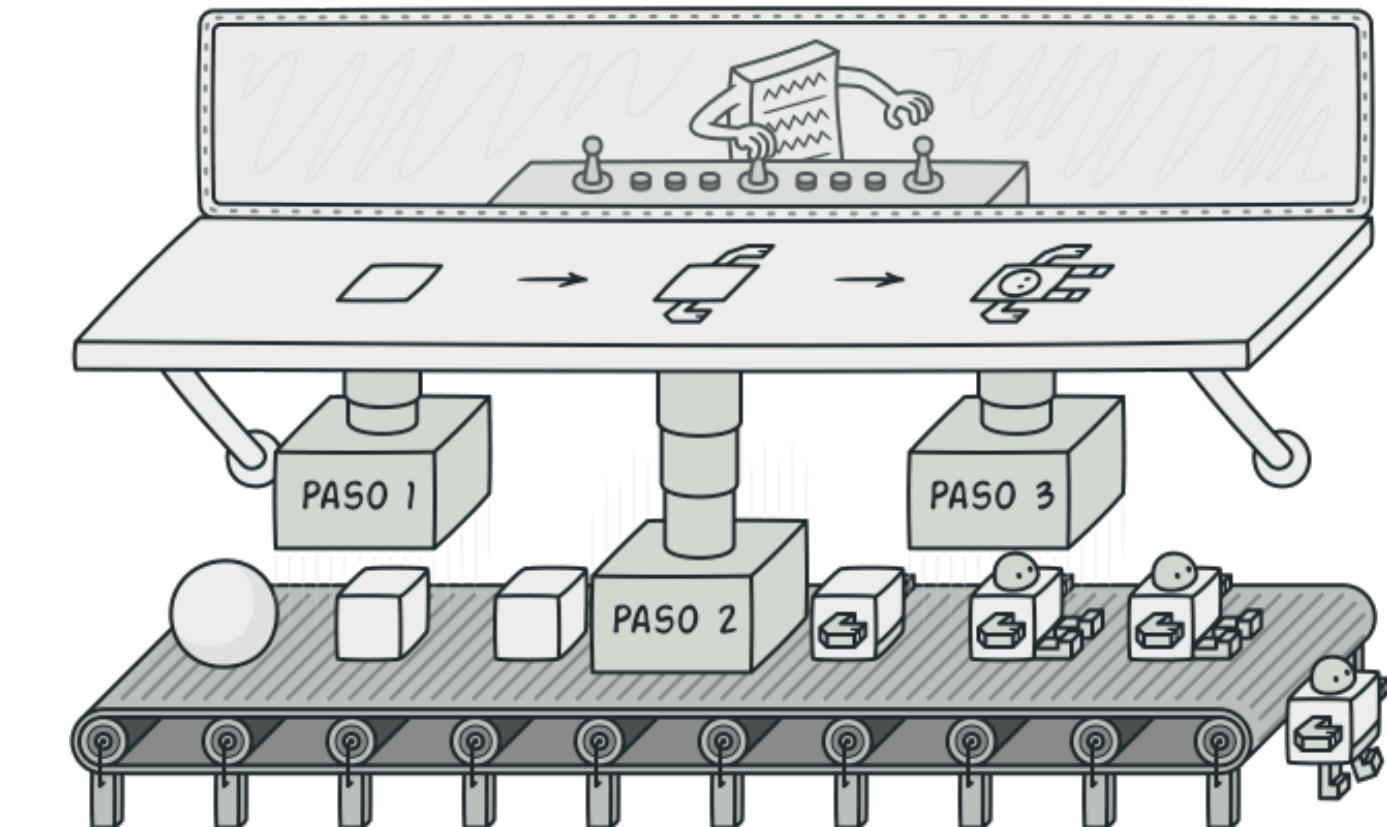
Patrones creacionales - Constructor



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.

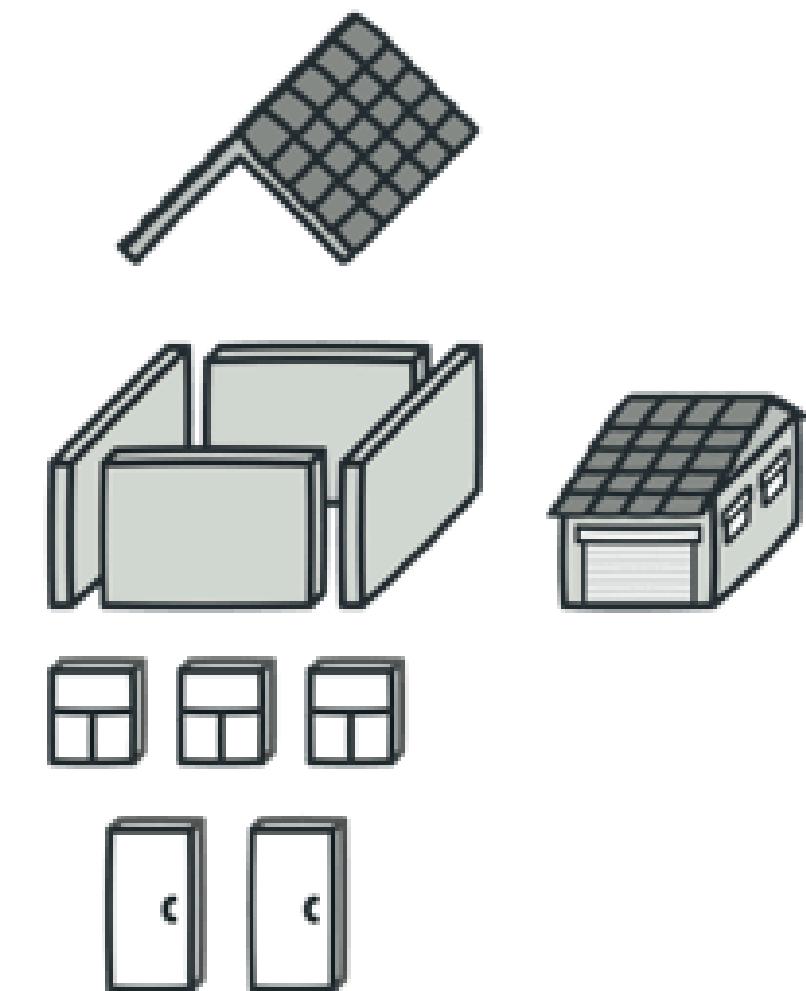
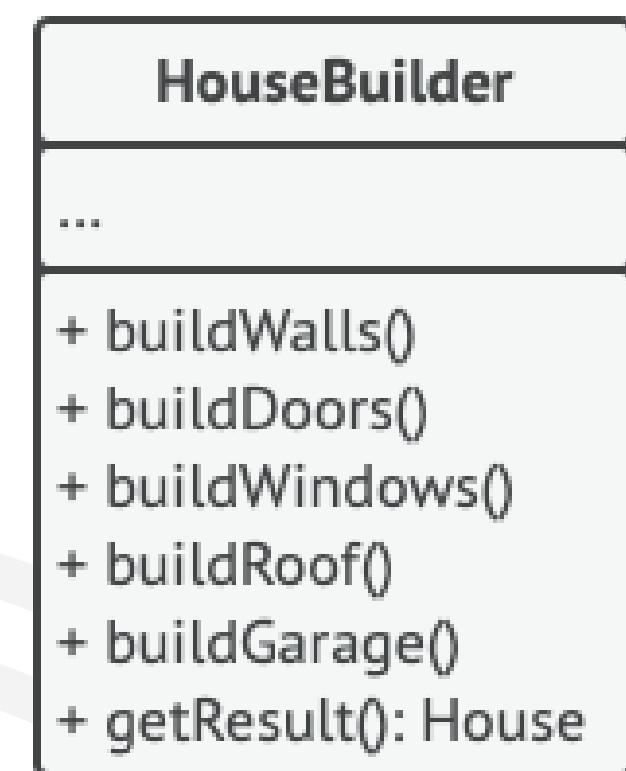


Patrones de Diseño - Builder

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.

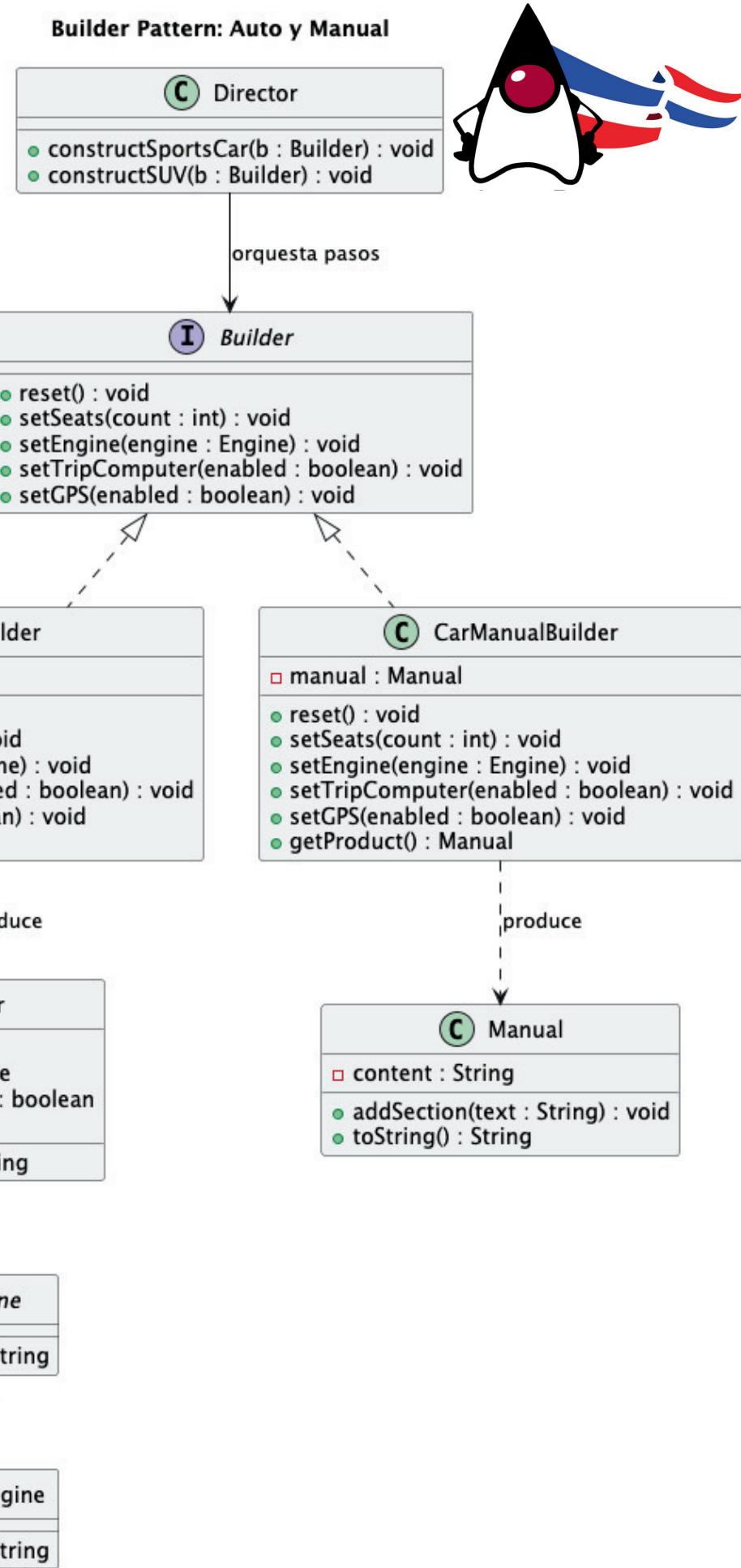


Patrones de Diseño - Builder

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

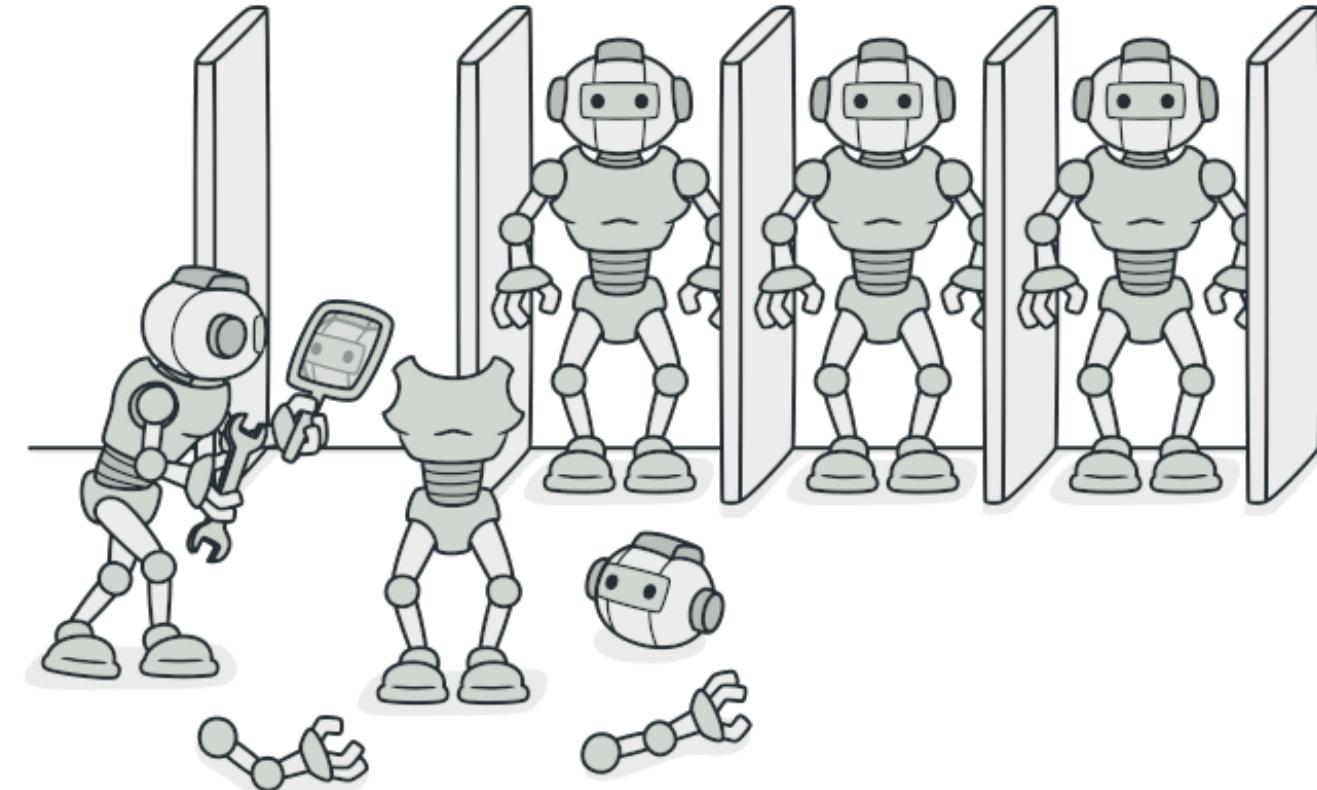
Patrones creacionales - Prototipo, Clone



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



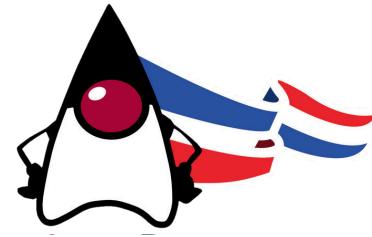
Aplicabilidad:

- 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.



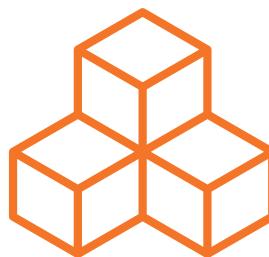
Pros y contras:

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

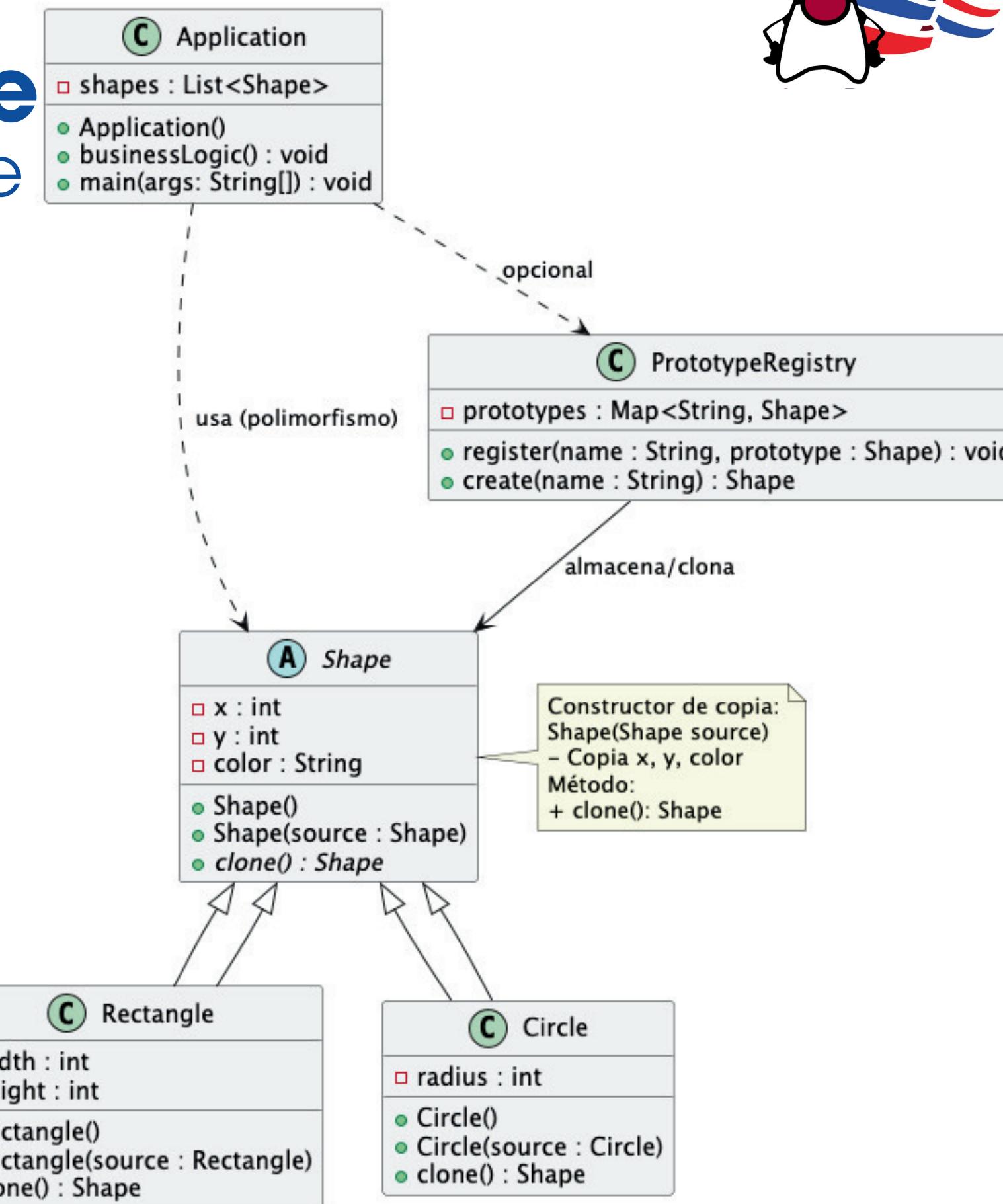


Patrones de Diseño - Prototype

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

Muchas Gracias

Reach out to me

-  freddy.pena@alphnology.com
-  github.com/fredpena
-  linkedin.com/in/fred-pena/
-  x.com/fred_pena

