UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

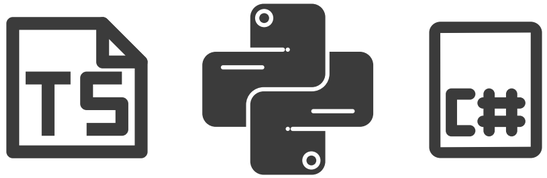
Paradigmas de Programación.

Existen muchos tipos de paradigmas de programación, y es muy fácil confundir los paradigmas con los lenguajes de programación.

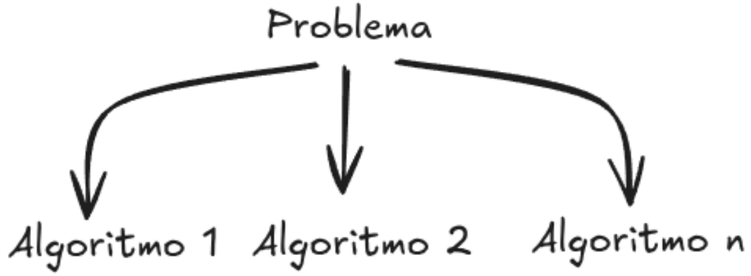
Los **lenguajes de programación** se utilizan para **“enseñar” a las computadoras a realizar diferentes tareas y acciones**, tal cual como los lenguajes que usamos para comunicarnos entre nosotros. Los lenguajes también tienen sus propios vocabularios y reglas gramaticales para desarrollar estas instrucciones.

Los **paradigmas** son **modelos de escritura de código que se pueden aplicar a varios lenguajes**. Incluso es posible utilizar más de un paradigma para la misma solución en el mismo lenguaje. Los paradigmas pueden entenderse como **un estilo o metodología de programación, que apuntan a la mejor manera de resolver problemas**.

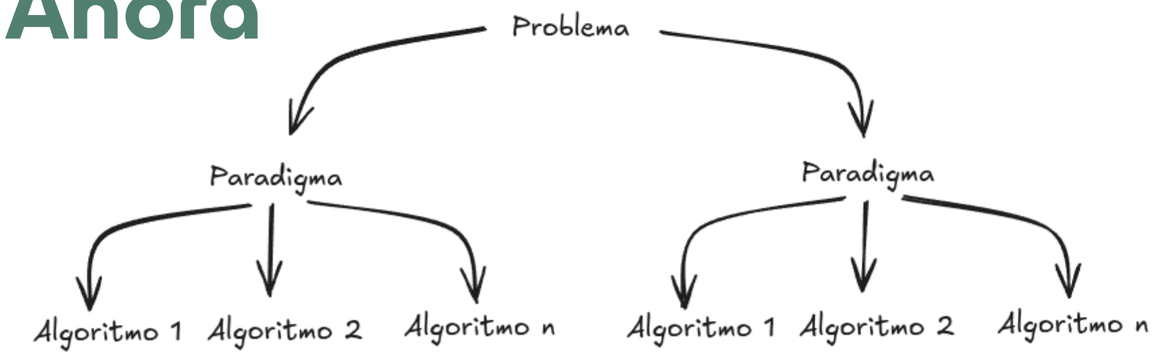
No hay un lenguaje que optimice todos los paradigmas a la vez.
Aunque... Muchos son multiparadigma



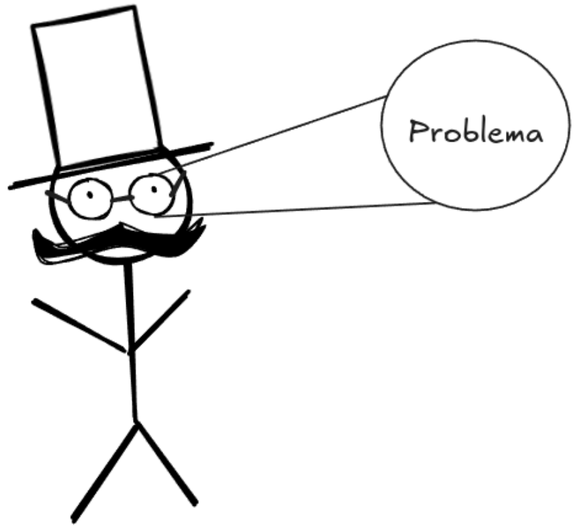
Antes



Ahora



Aprender paradigmas te da **técnicas transferibles**. Lo que aprendes resolviendo problemas en un estilo, puedes "*embeberlo*" en tu stack principal cuando lo necesites.



Paradigma Imperativo

Secuencia de instrucciones que se ejecutan en orden.

Es el **paradigma clásico**. Si te preguntan qué es un lenguaje de programación, es probable que la definición que des corresponda al paradigma imperativo.

Ejemplos de lenguajes: C, C++, Java, Python

Lo más importante del paradigma imperativo es el concepto de **estado**. Un programa imperativo tiene un estado (el conjunto de valores de todas sus variables en un momento dado) y consiste en una serie de comandos que **mutan ese estado** paso a paso para llegar al resultado final.

```
1  const numeros = [10, 20, 30, 40, 50];
2  let totalSuma: number = 0; // Inicialización del estado
3
4  // 1. Bucle 'for...of': una estructura de control."
5  for (const numero of numeros) {
6      // 2. Modificación del estado (mutación):
7      totalSuma = totalSuma + numero;
8  }
9
10 console.log(`La suma total es: ${totalSuma}`);
```

Paradigma Declarativo

*No definimos **cómo** se hacen las cosas, sino **qué** se hace.*

En palabras mas técnicas, expresiones sobre sentencias.

Una **sentencia** es una acción que se ejecuta y no devuelve un valor.
Podemos tomar como ejemplo: for, if, let x = 5

Una **expresión** es una pieza de código que se evalúa y **produce un valor**. En el mundo declarativo, los programas se construyen componiendo expresiones.
Podemos tomar como ejemplo: 2 + 3, miFuncion(x), x > 5

Ejemplos de lenguajes: SQL, HTML, CSS, FW como React

```sql
-- Declaramos QUÉ queremos obtener
SELECT titulo, anio_publicacion
FROM Libros
WHERE autor = 'Gabriel García Márquez';
```

Paradigma Funcional

Ninguna función puede modificar el estado de la aplicación
El programa es un **conjunto de funciones que al final devuelven un cálculo o un resultado**.

En este paradigma las funciones se conocen como **funciones puras**, y se caracterizan por:

1. No modifican el estado del programa.
2. Siempre devuelven el mismo valor dada la misma entrada.

Podemos tomar como ejemplos a:

- La función **suma(a, b)** dados los mismos **a** y **b**, siempre devolverá el mismo resultado.
- En cambio, la función **random()** que dada la misma entrada, puede devolver un número distinto.

¿Que les hace pensar esto?

Paradigma Funcional

Ninguna función puede modificar el estado de la aplicación
El programa es un **conjunto de funciones que al final devuelven un cálculo o un resultado**.

En este paradigma las funciones se conocen como **funciones puras**, y se caracterizan por:

1. No modifican el estado del programa.
2. Siempre devuelven el mismo valor dada la misma entrada.

Podemos tomar como ejemplos a:

- La función **suma(a, b)** dados los mismos **a** y **b**, siempre devolverá el mismo resultado.
- En cambio, la función **random()** que dada la misma entrada, puede devolver un número distinto.

Estas características hacen que el paradigma funcional se base fuertemente en la **recursion**. Y lo mas importante de todo: **fácil de debuggear**.

Caracteristicas

- **Función de Orden Superior**: Funciones que toman a otra función como argumento.
- **Inmutabilidad**: Estas funciones no deben modificar la estructura de datos original. En su lugar devuelven **nuevos datos** con los elementos transformados.
- **Ausencia de Efectos Colaterales**: Funciones no producirán ningún efecto colateral. Solo transforma datos y devuelve un nuevo valor, sin alterar ningún estado externo.
- **Estilo Declarativo**: Promueve un estilo de programación declarativo.

Lambdas (Arrow Functions) λ

Caracteristicas

- **Función de Orden Superior**: Funciones que toman a otra función como argumento.
- **Inmutabilidad**: Estas funciones no deben modificar la estructura de datos original. En su lugar devuelven **nuevos datos** con los elementos transformados.
- **Ausencia de Efectos Colaterales**: Funciones no producirán ningún efecto colateral. Solo transforma datos y devuelve un nuevo valor, sin alterar ningún estado externo.
- **Estilo Declarativo**: Promueve un estilo de programación declarativo.

Lambdas (Arrow Functions) λ

Son funciones anónimas que normalmente se definen “en línea”

Propiedades:

- Son concisas y se usan inline.
- No tienen su propio contexto (this/arguments)
- Pueden ser puras o impuras, dependiendo de si usan o modifican estado externo.
- Permiten crear closures: funciones que “recuerdan” variables del contexto donde se crearon.

Paradigma Reactivo

P. de programación asíncrona que gira alrededor de los flujos de datos y la propagación de cambios.

Incluso existe un **manifiesto**: *Queremos sistemas responsivos, Elásticos y Orientados a Mensajes.*

El ejemplo clásico son las hojas de excel.

ó

```
x = <mouse-x>;
y = <mouse-y>;
```

Y si tenemos: a = x + y

¿Que pasaría en un paradigma tradicional?

¿Que pasaría en el paradigma reactivo?

¿Saben en qué casos esto es muy útil? Cuando debemos **reaccionar** a distintas interacciones.

(Tal vez conozcan el patrón de diseño **Observador**)

Pero cuidado, se puede complicar...

Orientado a Objetos

Concibe el sistema como un conjunto de entidades (los objetos) que modelan elementos del mundo real; cada objeto encapsula su información y su comportamiento, y cooperan entre sí enviándose mensajes para conseguir un objetivo común.

Esto pretende:

- Construir componentes de software que sean **re utilizables**.
- Diseñar soluciones de manera que puedan ser **extendidas y modificadas** con el mínimo impacto en el resto de su estructura.

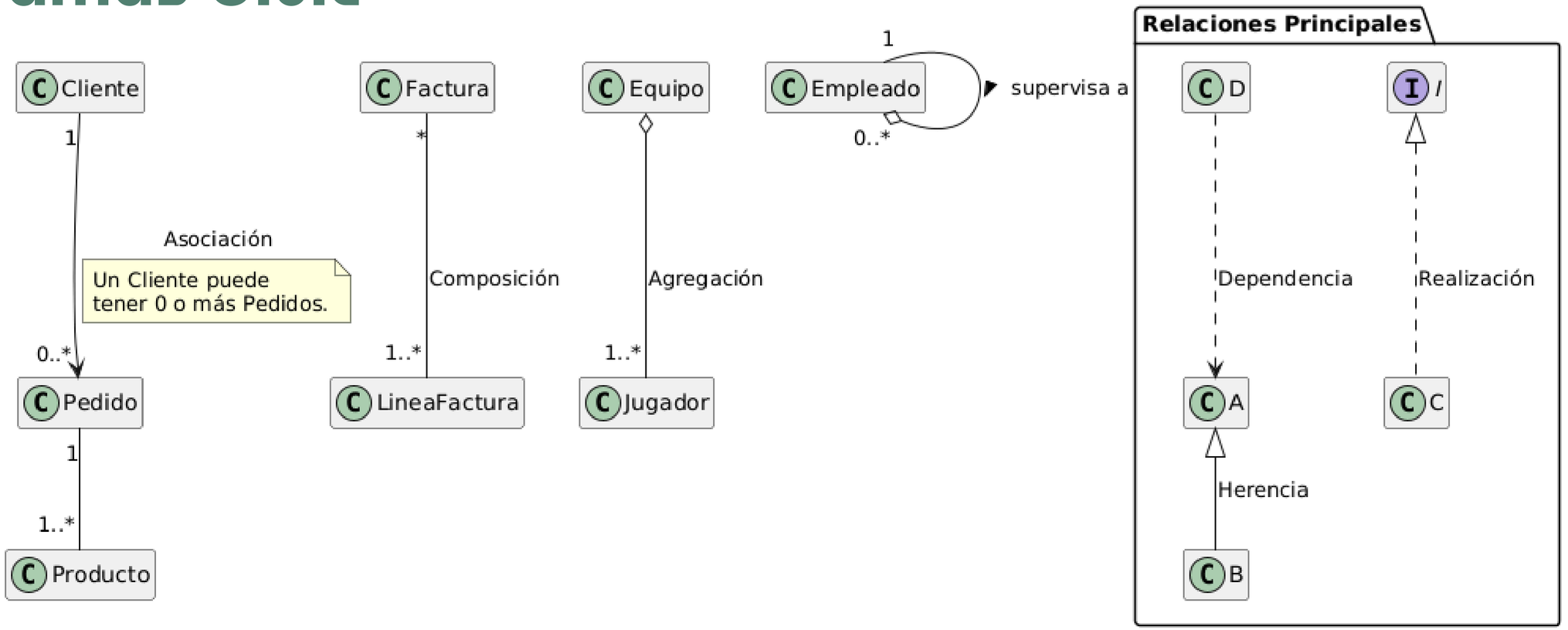
Todo objeto tiene, dispone o conoce:

1. **Un estado interno**, conformado por atributos o variables de instancia que describen cómo es la entidad y que contienen los valores que representan su información.
2. **El comportamiento** (es el qué), que consiste en el conjunto de mensajes que puede recibir, lo que se corresponde con las acciones o funcionalidades que puede realizar la entidad. Son la única via para comunicarse con los objetos.
3. **Métodos** (es el como). Los métodos corresponden a las implementaciones de los comportamientos de los objetos. Los métodos se ejecutan en respuesta a un mensaje
4. Todo objeto posee una **identidad**.

Tenemos:

- **Atributos**: Son los datos que un objeto guarda sobre sí mismo. Son las características que lo describen y lo hacen único.
- **Métodos**: Son las acciones o habilidades que un objeto sabe realizar. Son las instrucciones que se ejecutan cuando el objeto recibe un mensaje.
- **Clase**: Es el **molde** que usamos para crear objetos. Define la estructura común que todos los objetos de un mismo tipo compartirán.

Diagramas UML



Asociación: "usa un"/"tiene un"/"referencia". Los objetos de dos clases se relacionan entre sí.

Composición: El "todo" es dueño de las "partes". Si el "todo" se destruye, las partes también.

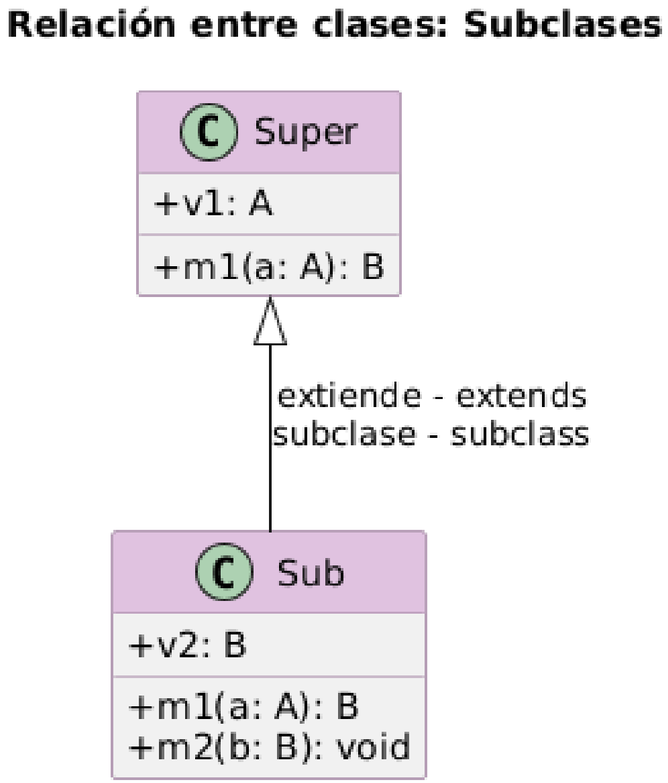
Agregación: "todo/parte". El "todo" tiene "partes", pero las partes pueden existir sin el todo.

Dependencia: Una clase "depende" de otra si la usa, por ejemplo, como un parámetro en un método.

Herencia: "es un". Una subclase hereda todo de su superclase

Realización: "implementa un". Una clase provee el código para los métodos definidos en una interfaz.

Herencia



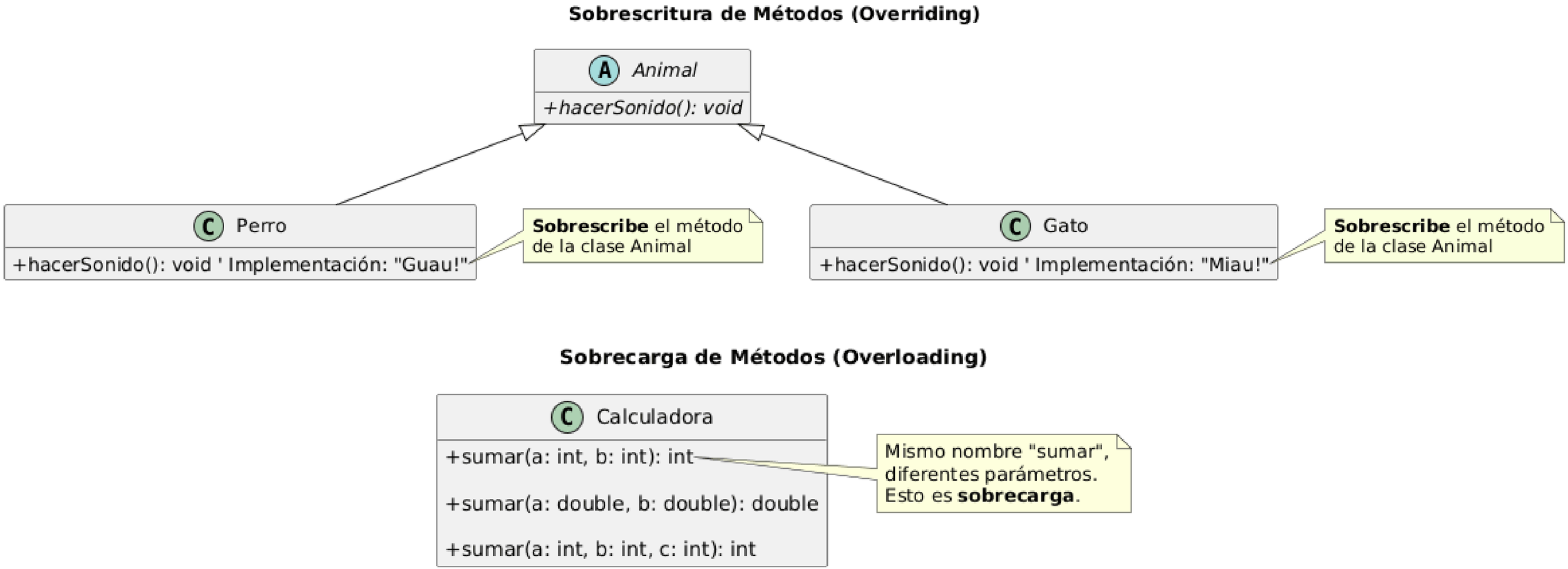
Es uno de los pilares de la **POO**, nos permite crear una jerarquía de clases, donde una clase más específica (la **subclase**) hereda características de una clase más general (la **superclase**).

Ventajas:

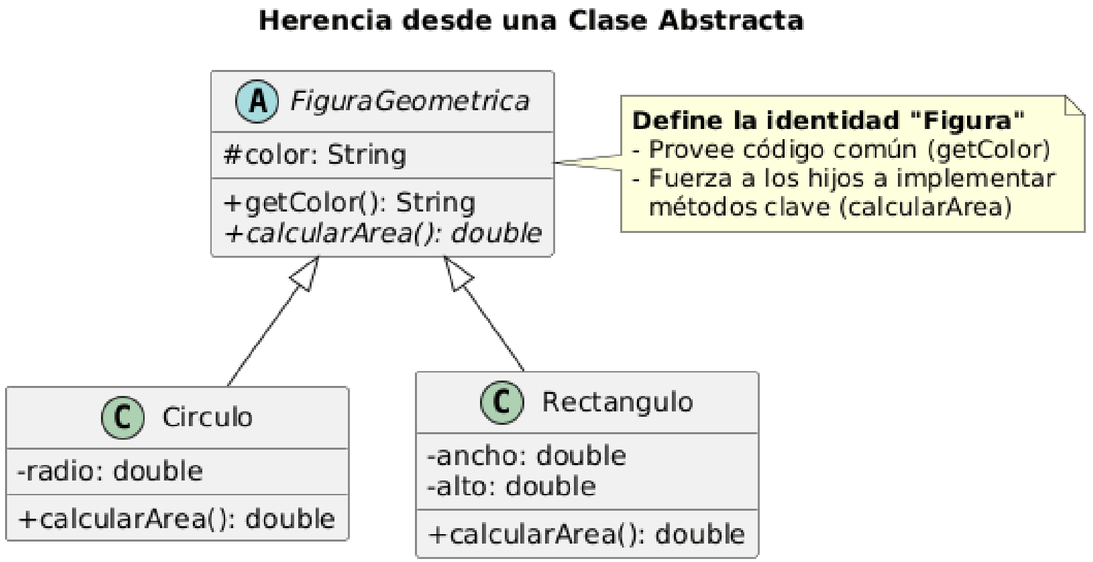
1. R**eutilización de Código**.
2. **Organización Lógica**. Modelando el mundo real de una forma muy natural.
3. **Polimorfismo**. Permitiendo tratar a objetos de las subclases como si fueran objetos de la superclase.

Una subclase puede **agregar nuevos métodos**. Pero también puede **sobreescribir** un método de la superclase

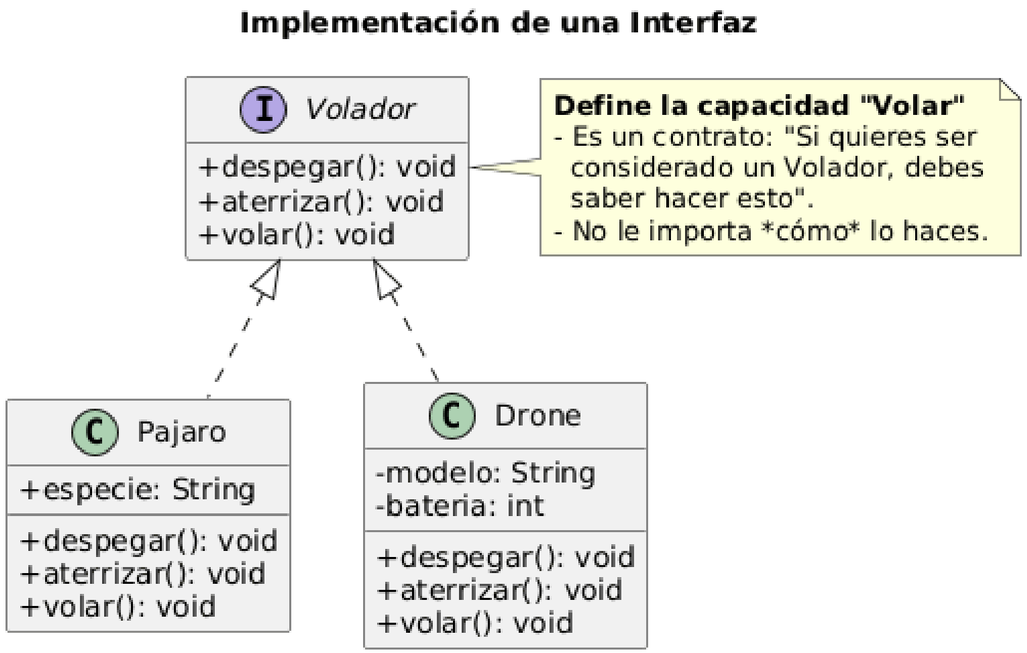
Sobreescritura y Sobrecarga



Clases Abstractas



Interfaces



Fin