La fase de análisis II: Del análisis al diseño orientado a objetos

Una vez que sabemos qué debe hacer el sistema, debemos decidir cómo lo va a hacer. El diseño orientado a objetos nos permite organizar el sistema en clases, atributos, métodos y relaciones entre objetos. Esta forma de estructurar el software lo hace más mantenible, reutilizable y cercano a la realidad.

En esta unidad vamos a aprender a:

- Identificar clases, atributos y métodos a partir de los requisitos
- Diseñar relaciones entre clases (herencia, asociación, composición…)
- Representar el diseño con diagramas de clases
- Aplicar principios básicos del diseño orientado a objetos

> **Actividad**
> Reflexiona sobre la siguiente pregunta:
>
> ¿Por qué crees que es importante organizar bien el código antes de empezar a programar?
>
> Después, debate en clase los siguientes puntos:
>
> - ¿Qué ventajas ofrece el enfoque orientado a objetos frente a otros estilos de programación?
> - ¿Crees que entender cómo se relacionan las clases ayuda a programar más rápido o más seguro? ¿Por qué?
> - ¿Qué papel juega el diseño en el trabajo en equipo?
> - ¿Un buen diseño puede evitar errores durante la implementación? ¿Cómo?

El paradigma orientado a objetos

Programación Orientada a Objetos (POO): Conceptos Fundamentales

El paradigma de la **Programación Orientada a Objetos (POO)**, del inglés *Object-Oriented Programming (OOP)*, es un enfoque de programación que se fundamenta en el concepto de **objeto**. Un objeto es una entidad que encapsula datos y los procedimientos que operan sobre esos datos, denominados **métodos**, en una única unidad cohesionada.

Distinción con Paradigmas Imperativos

A diferencia de los paradigmas imperativos, donde los datos se almacenan en variables y su manipulación se realiza mediante funciones independientes, la POO integra la información y el comportamiento dentro de la misma estructura del objeto. Esta integración eleva el nivel de abstracción y favorece la modularidad en el desarrollo de software.

Principios Clave de la POO

La POO se asienta sobre pilares conceptuales que facilitan el diseño, la implementación y el mantenimiento de sistemas complejos:

- **Encapsulación:** Este principio consiste en la agrupación de los datos (atributos) y los métodos que operan sobre ellos dentro de una única unidad, el objeto. La encapsulación restringe el

acceso directo a los datos internos de un objeto, obligando a la interacción a través de una **Interfaz de Programación de Aplicaciones (API)** pública o conjunto de métodos expuestos. Esto protege la integridad de los datos, previene modificaciones inconsistentes y facilita la evolución del código sin afectar a otras partes del sistema que interactúan con el objeto.

- **Abstracción:** Implica la representación simplificada de entidades del mundo real o conceptos complejos, enfocándose en sus características esenciales y ocultando los detalles de implementación subyacentes. La POO permite a los desarrolladores trabajar con objetos a un nivel conceptual, sin necesidad de comprender su funcionamiento interno detallado, facilitando así la gestión de la complejidad.
- **Polimorfismo:** Del griego "muchas formas", el polimorfismo permite que un mismo nombre de método o mensaje pueda invocar diferentes implementaciones según el tipo específico de objeto que lo reciba. Esto conduce a un código más flexible, extensible y reutilizable, ya que se puede interactuar con diferentes objetos de manera uniforme a través de una interfaz común.
- **Herencia:** En sistemas basados en **clases**, la herencia es un mecanismo que permite a una clase (subclase o clase derivada) adquirir propiedades y comportamientos (atributos y métodos) de otra clase (superclase o clase base). Este principio promueve la **reutilización de código** y la organización jerárquica de los objetos, modelando relaciones "es un tipo de" (por ejemplo, un "Coche" es un tipo de "Vehículo").

Clases y Prototipos

La creación y estructuración de objetos en la POO se puede abordar de dos formas principales:

- **Clases:** Son plantillas o moldes que definen la estructura y el comportamiento común de un conjunto de objetos. Un objeto creado a partir de una clase se denomina **instancia** de esa clase. La mayoría de los lenguajes de POO (como Java, C++, Python, C#) se basan en clases y en el concepto de herencia para la creación de jerarquías de tipos.
- **Prototipos:** En algunos lenguajes (notablemente JavaScript), los objetos se crean directamente a partir de otros objetos existentes, denominados prototipos. En este modelo, un objeto puede servir como plantilla para la creación de nuevos objetos, los cuales heredan directamente las propiedades y métodos del prototipo.

> **Actividad**
> De los siguientes ejemplos del mundo real, separa los datos de los métodos. Esta habilidad es crucial a la hora de modelar objetos a partir de especificaciones.
>
> 1. Un Turismo (Coche)
> 2. Un Tucán (pájaro)
> 3. Una profesora de informática (Persona)

Principios SOLID

Los **principios SOLID** son un conjunto de cinco principios de diseño de software que tienen como objetivo hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener. Fueron introducidos por Robert C. Martin, también conocido como "Uncle Bob", y son una guía fundamental en la programación orientada a objetos para escribir código limpio y robusto. Al aplicarlos, se busca crear sistemas que sean más resistentes a los cambios y menos propensos a errores.

S - Principio de Responsabilidad Única (Single Responsibility Principle - SRP)

El SRP establece que **una clase debe tener una, y solo una, razón para cambiar**. Esto significa que una clase debe ser responsable de una única funcionalidad o de un único aspecto de la aplicación. Si una clase tiene múltiples responsabilidades, cualquier cambio en una de esas responsabilidades podría afectar a las demás, haciendo que el código sea más frágil y difícil de mantener.

Ejemplo: En lugar de tener una clase `Reporte` que se encargue tanto de generar los datos del informe como de formatearlos para su impresión, sería mejor tener dos clases separadas: una `GeneradorDeDatosDeReporte` y una `FormateadorDeReporte`. Así, si cambian las reglas de negocio para los datos, solo se modifica la primera; si cambia el diseño de impresión, solo se modifica la segunda.

O - Principio de Abierto/Cerrado (Open/Closed Principle - OCP)

El OCP dicta que **las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas a la extensión, pero cerradas a la modificación**. Esto significa que el comportamiento de un módulo puede ser extendido sin necesidad de alterar su código fuente existente. En esencia, una vez que una clase ha sido probada y liberada, no deberíamos tener que modificarla para añadir nuevas funcionalidades.

Ejemplo: Si tienes una clase `CalculadoraArea` que calcula el área de diferentes formas geométricas, en lugar de añadir nuevos `if/else` o `switch` dentro de ella cada vez que aparece una nueva forma (lo que implicaría modificarla), el OCP sugiere que se debería extender su funcionalidad a través de la herencia o la implementación de interfaces. Podrías tener una interfaz `Forma` con un método `calcularArea()`, y luego clases como `Circulo` o `Rectangulo` que implementen esa interfaz. La `CalculadoraArea` entonces operaría sobre la interfaz `Forma`.

L - Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)

El LSP, formulado por Barbara Liskov, afirma que **los objetos de un programa deben ser reemplazables por instancias de sus subtipos sin alterar la corrección de ese programa**. En términos más simples, si tienes una clase `B` que hereda de una clase `A`, entonces `B` debe poder usarse en cualquier lugar donde se espere `A` sin causar problemas. Esto implica que los subtipos no deben cambiar el comportamiento esperado definido por sus supertipos.

Ejemplo: Si tienes una clase `Pájaro` con un método `volar()`, y luego creas una subclase `Pingüino` que también hereda de `Pájaro`, pero los pingüinos no vuelan. Si un código espera un `Pájaro` y llama a `volar()`, y se le pasa un `Pingüino`, el programa podría comportarse de manera inesperada o lanzar un error. El LSP sugiere que `Pingüino` no debería heredar directamente de `Pájaro` si no cumple con el contrato de comportamiento de volar, o que la jerarquía debería rediseñarse (por ejemplo, tener una clase `AnimalQueVuela` y otra `AnimalQueNoVuela`).

I - Principio de Segregación de la Interfaz (Interface Segregation Principle - ISP)

El ISP establece que **los clientes no deberían ser forzados a depender de interfaces que no utilizan**. Es decir, es mejor tener muchas interfaces pequeñas y específicas que una interfaz grande y monolítica. Una interfaz grande podría obligar a las clases a implementar métodos que no necesitan, lo que rompe el Principio de Responsabilidad Única y dificulta el mantenimiento.

Ejemplo: Imagina una interfaz `Trabajador` con métodos `trabajar()`, `comer()`, `dormir()`, `cobrarSueldo()`. Si tienes un `RobotTrabajador` que no necesita `comer()` o `dormir()`, al implementar `Trabajador`, se vería obligado a implementar esos métodos, quizás vacíos o lanzando excepciones. El ISP sugiere dividir `Trabajador` en interfaces más pequeñas como `TrabajadorActivo` (con `trabajar()`), `SerVivo` (con `comer()`, `dormir()`) y `Asalariado` (con `cobrarSueldo()`). Así, `RobotTrabajador` solo implementaría `TrabajadorActivo`.

D - Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP)

El DIP establece que:

1. **Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.**

2. **Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.**

En esencia, este principio promueve el uso de abstracciones (interfaces o clases abstractas) para acoplar las clases, en lugar de depender de implementaciones concretas. Esto reduce el acoplamiento entre los módulos y facilita la flexibilidad y la facilidad de prueba.

Ejemplo: Si una clase `ProcesadorDePedidos` depende directamente de una implementación concreta de `BaseDeDatosSQL` para guardar los pedidos, está fuertemente acoplada a ella. Si se quiere cambiar a una `BaseDeDatosNoSQL`, habría que modificar `ProcesadorDePedidos`. El DIP sugiere que `ProcesadorDePedidos` dependa de una interfaz `RepositorioDePedidos` (una abstracción), y `BaseDeDatosSQL` (o `BaseDeDatosNoSQL`) implementen esa interfaz. Así, `ProcesadorDePedidos` no necesita saber los detalles de la base de datos subyacente.

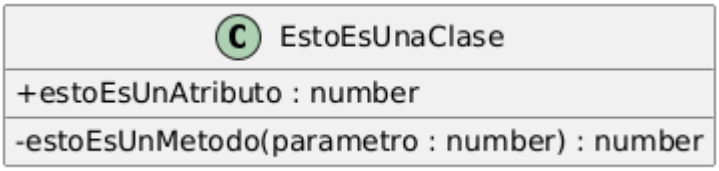
Aplicar los principios SOLID puede parecer un esfuerzo inicial, pero a largo plazo, resultan en sistemas más robustos, escalables y fáciles de mantener, lo que es crucial en el desarrollo de software moderno.

Diagramas de clase UML

Construcción de un diagrama de clases

Para representar una clase emplearemos diagramas de clases del estándar UML. Cada clase se representa con una caja que tiene tres apartados:

- **Superior**: Se coloca el nombre de la clase y su tipo. Puede ser una clase normal (C), una clase abstracta (A o el nombre en cursiva) o una interfaz (I o añadiendo `<<interface>>` al nombre de la clase). En Java, se emplea CamelCase para los nombres. Es decir, la primera letra de cada palabra en mayúscula y sin espacios. Se evitan caracteres especiales que no pertenezcan al ANSII (como tildes, ñ o ç).
- **Medio**: En este espacio se colocan los atributos de la clase, en CamelCase pero con la primera letra en minúscula (lowerCamelCase). El tipo se puede colocar antes, como en Java o C#, o después, como en TypeScript.
- **Inferior**: En este espacio se colocan los métodos. Se nombran como los atributos, pero al final tienen paréntesis. Dentro del paréntesis puede haber parámetros (con su tipo correspondiente) y, si el método devuelve algo, se coloca al lado el tipo de datos del retorno.



▶ **Haz click aquí para ver el código plantuml**

Métodos y atributos

Podemos pensar en los atributos y métodos de una clase de la siguiente forma:

- **Atributos:** Conforman las propiedades del objeto.
- **Métodos:** Conforman el comportamiento del objeto.

Tipos de visibilidad

Cuando definimos una clase, los diferentes componentes tienen una visibilidad determinada:

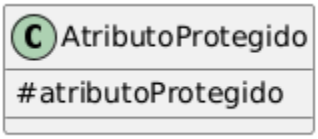
- Si un componente es **público**, es accesible por cualquier otro objeto. Lo representamos con un +:

▶ **Haz click aquí para ver el código plantuml**

- Si un componente es **privado**, es accesible solo por el mismo objeto. Lo representamos con un -:

▶ **Haz click aquí para ver el código plantuml**

- Si un componente es **protegido**, funciona como privado excepto cuando el objeto que quiere acceder a él es de una clase derivada, que entonces funciona como público. Lo representamos con un #:



▶ **Haz click aquí para ver el código plantuml**

- Si un componente tiene visibilidad de **paquete**, funciona como público para todos los miembros del paquete y privado para todos los demás. Es una visibilidad característica del lenguaje de programación Java (es la visibilidad por defecto, de hecho). En UML se representa con el signo ~:

▶ **Haz click aquí para ver el código plantuml**

La visibilidad es una herramienta clave a la hora de determinar la **encapsulación** de la información. A las partes públicas (generalmente métodos) se las conoce como la API (application programming interface) del objeto.

Herramientas para la creación de Diagramas de clases UML

Para estos diagramas se está usando PlantUML. Para que aparezcan los signos indicados se debe especificar antes de empezar a escribir el diagrama la línea `skinparam classAttributeIconSize 0`. Si no se hace, aparecen formas geométricas de diferentes colores según la visibilidad, y están rellenas o no según si son atributo o método:

▶ **Haz click aquí para ver el código**

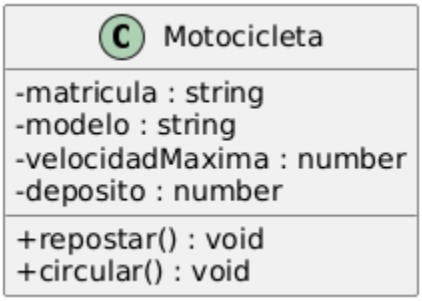
Además de Plantuml, puedes usar Mermaidjs. Tanto Plantuml como Mermaid utilizan un lenguaje que es muy similar a un lenguaje de programación orientado a objetos convencional.

Por otro lado, existen herramientas especializadas como Visual Paradigm Online, Lucidchart o Miro que sirven para crear diagramas de clases UML (y otros muchos diagramas). Técnicamente, puedes emplear cualquier herramienta de diseño para crear un diagrama de cualquier tipo.

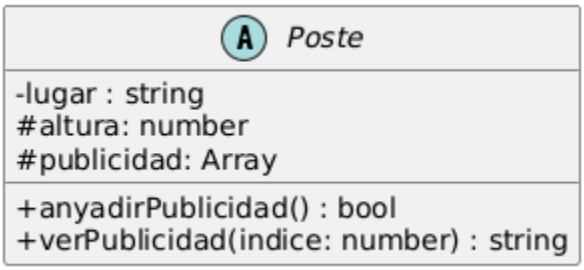
> **Actividad**
> Crea un diagrama de clase que se corresponda con cada una de estas definiciones:
>
> - Para representar un libro, necesitamos saber su autor, su editorial, su año de publicación, su ISBN y su número de páginas. Un libro se puede leer.

> - Para representar a un perro, necesitamos saber su nombre, su raza y su edad. Un perro puede ladrar y pasear.

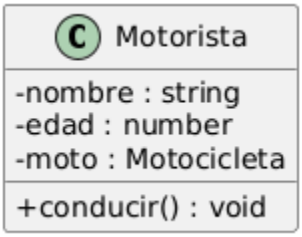
> **Actividad**
>
> Explica los siguientes diagramas:



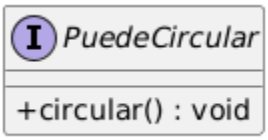
▶ **Haz click aquí para ver el código plantuml**



▶ **Haz click aquí para ver el código plantuml**



▶ **Haz click aquí para ver el código plantuml**



▶ **Haz click aquí para ver el código plantuml**

Modelización de Clases y Objetos

Continuando con la Programación Orientada a Objetos, una vez comprendida la estructura de una **clase** y un **objeto**, el siguiente paso fundamental es cómo los modelamos para que representen eficazmente el mundo real o el problema que queremos resolver. La **modelización** es el proceso de identificar las clases y objetos necesarios, definir sus atributos y comportamientos, y establecer las relaciones entre ellos, aún sin introducir la herencia.

Identificación de Clases y Objetos

El primer paso en la modelización es identificar las entidades clave en el dominio del problema. El dominio del problema se establece en el documento de Especificación de Requerimientos del Sistema (IEEE 830) o en documentos similares. Generalmente, las **clases** corresponden a sustantivos o conceptos abstractos importantes. Es muy similar al diagrama Entidad-Relación que se trabaja en bases de datos, pero en el diagrama de clases UML también modelizamos las acciones que realiza cada elemento.

- **Clase:** Representa una categoría o un "plano" general para un conjunto de objetos que comparten características y comportamientos comunes.
  - **Ejemplos:** `Coche`, `Libro`, `Cliente`, `Factura`, `Producto`.
- **Objeto (Instancia):** Es una ocurrencia concreta de una clase, con valores específicos para sus atributos.
  - **Ejemplos:** Un objeto `Coche` con color "rojo", marca "Ford", modelo "Fiesta". Un objeto `Libro` con título "Cien años de soledad" y autor "Gabriel García Márquez".

¿Cómo identificamos clases?

1. **Sustantivos:** Los sustantivos en la descripción del problema suelen ser buenos candidatos para clases.
2. **Entidades tangibles:** Personas, lugares, cosas.
3. **Conceptos abstractos:** Eventos, transacciones, roles.

Definición de Atributos y Métodos

Una vez identificadas las clases, debemos determinar qué información almacenará cada objeto (sus **atributos**) y qué acciones podrá realizar (sus **métodos**).

- **Atributos:** Son las propiedades o características que describen el estado de un objeto. Cada objeto de la misma clase tendrá los mismos atributos, pero con valores potencialmente diferentes.
  - **Ejemplo para la clase `Coche`:** `matricula` (string), `modelo` (string), `color` (string), `velocidadActual` (number).
- **Métodos:** Representan las operaciones o comportamientos que un objeto puede llevar a cabo o que se pueden realizar sobre él. Los métodos suelen ser verbos o frases verbales.
  - **Ejemplo para la clase `Coche`:** `arrancar()` (void), `acelerar(incremento: number)` (void), `frenar()` (void), `obtenerVelocidad()` (number).

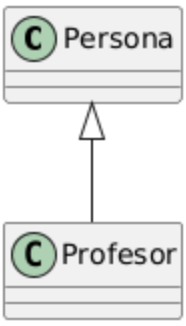
> **Actividad**
> Extrae y modela a partir del siguiente enunciado las posibles clases:
>
> Se nos pide un sistema para gestionar la entrega de diplomas de un instituto. Los diplomas se entregan a alumnos, identificados por su nombre y NIA, que cursan asignaturas, que se identifican por el nombre de la asignatura, su código y su curso académico. En un diploma se recoge la información básica sobre la asignatura y el alumno, añadiendo además una calificación. Un alumno puede matricularse en una asignatura y también darse de baja. Una asignatura puede ser modificada. Un diploma puede expedirse.

Relaciones entre clases

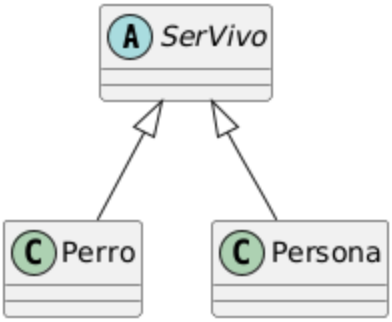
Relaciones de Herencia

A través de los diagramas UML podemos expresar las relaciones entre las clases de múltiples maneras, mediante líneas que las unen. A su vez, también podemos especificar cardinalidades, como en los diagramas Entidad-Relación empleados en base de datos, para cuantificar los detalles de las relaciones, e incluso nombrar a dichas relaciones (generalmente con un verbo). Representamos las cardinalidades con 0, 1, muchos (representado a veces con $*$ o una letra) o números fijos. Las clases se pueden relacionar entre ellas de las siguientes maneras:

- **Herencia:** "es un" (relación jerárquica). Una clase deriva de otra. La clase base se denomina a veces clase padre y la clase derivda clase hija. En algunos casos, la clase padre es una clase abstracta. Esto significa que la clase abstracta no puede ser instanciada, pero las derivadas (mientras no sean abstractas a su vez) sí. Las clases abstractas, a la hora de ser programadas, pueden tener métodos abstractos, que son métodos sin definir (se definen en las clases derivadas). Más adelante veremos que hay herencia simple y múltiple, así como otros tipos de herencia.



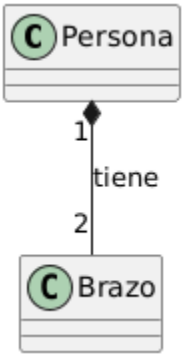
▶ **Haz click aquí para ver el código plantuml**



▶ **Haz click aquí para ver el código plantuml**

Relaciones de componente

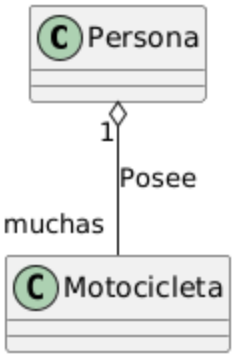
- **Componente:** "tiene un". Una clase contiene como atributo un objeto (o varios) de otra.
- **Composición:** De tipo *componente*. En este caso, el atributo es dependiente de la clase principal.



▶ **Haz click aquí para ver el código plantuml**

Una persona tiene dos brazos. Los brazos no pueden existir sin la persona.

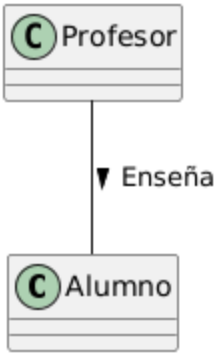
- **Agregación:** De tipo *componente*. En este caso, el atributo no es dependiente de la clase principal y puede existir por separado.



▶ **Haz click aquí para ver el código plantuml**

Una persona posee muchas motocicletas. Cada motocicleta puede existir sin la persona.

- **Asociación:** "colabora con". Dos clases pueden trabajar en colaboración. La colaboración puede ser simétrica o asimétrica (señalamos la dirección con una flecha). Aunque no es exactamente el mismo concepto, la asociación se puede usar de forma alternativa a la agregación y la composición. De hecho, muchas veces, el resultado al programarla será el mismo.

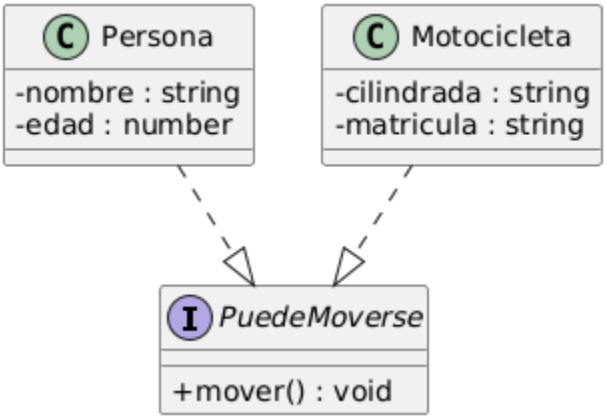


▶ **Haz click aquí para ver el código plantuml**

Otras relaciones

Además de esas relaciones básicas, existen otras un poco más avanzadas.

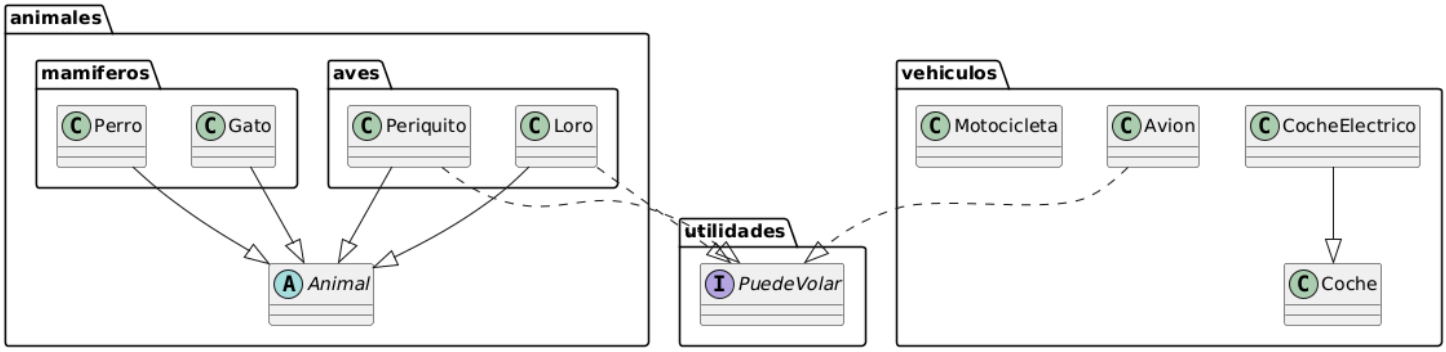
- **Dependencia:** "usa". En este tipo de relación, se dice que una clase usa a otra. Este uso puede ser mediante clases que contengan otras clases o mediante la implementación de interfaces dentro de una clase. Se representa como la herencia, pero con línea discontinua. A efectos de programación, una interfaz y una clase abstracta son muy similares. En ambos casos, no permiten instancias de ellas, sino de clases derivadas o que las usen. En Java, las interfaces vienen a compensar las limitaciones que tiene el hecho de que no exista la herencia múltiple.



▶ **Haz clik aquí para verl el código plantuml**

Un uso muy habitual de las interfaces es el de agrupar instancias que semánticamente son cosas distintas, pero que todas pueden realizar la misma acción. De esta forma, les podemos pedir a todas ellas con un mensaje que realicen dicha acción, aunque sean cosas tan dispares como una puerta o un plazo de entrega (en ambos casos, se podrían cerrar).

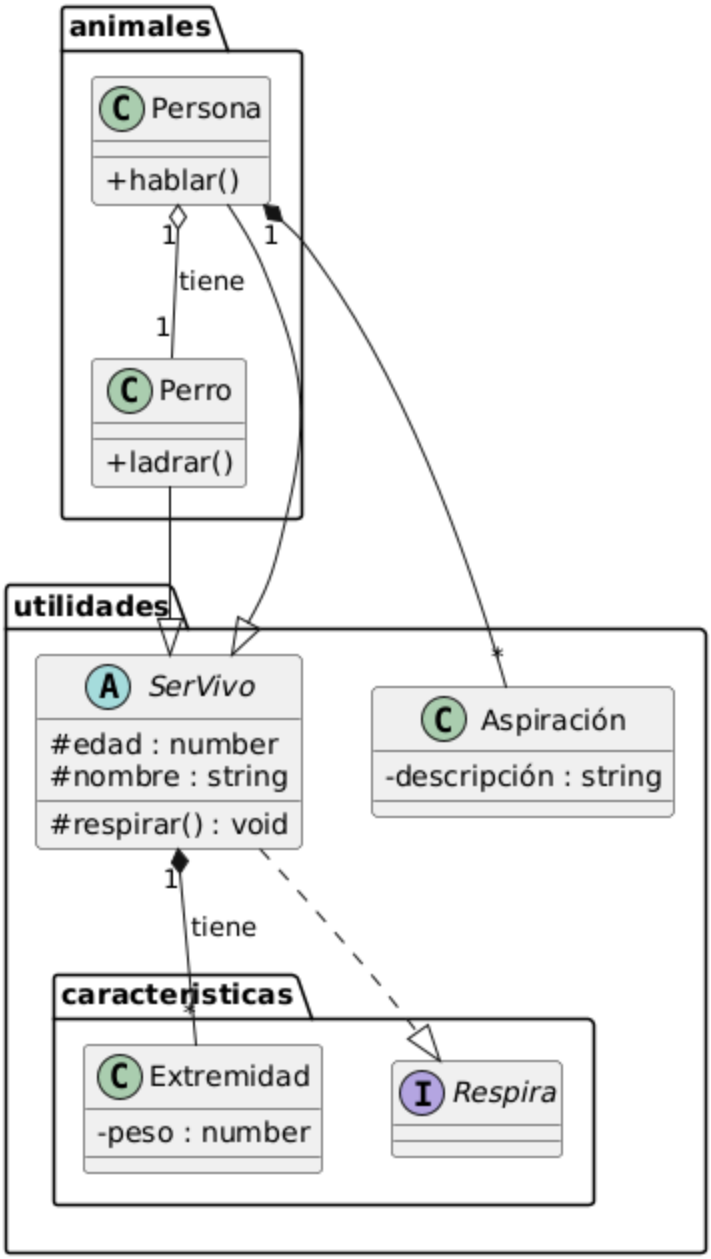
- **Pertenencia al mismo paquete:** En caso de que queramos especificar también los paquetes, podemos encapsular las clases en formas geométricas para verlo claro, de la siguiente manera:



▶ **Haz click aquí para ver el código plantuml**

Como se puede observar, puede haber paquetes dentro de paquetes y las clases de un paquete pueden relacionarse con clases de otros paquetes. Los paquetes tienen una relación más de conveniencia a la hora de programar que semántica.

> **Actividad**
> Explica el siguiente diagrama:



▶ **Haz click aquí para ver el código plantuml**

> **Actividad**
>
> Crea un diagrama de la siguiente explicación
>
> La vida es dura, pero la vida de cada persona tiene un grado de dureza diferente, que clasificamos con un número. Las personas asismo tienen algo que las identifica, su propio nombre. Pueden elegir varias profesiones, como carpintero o influencer. Los carpinteros usan la madera para construir muebles. Los influencer usan las mesas, que son muebles, para poner sus ordenadores. Una persona puede tener varios muebles. Las sillas también son muebles.

Técnicas de modelado

Modelizar las relaciones de Herencia y sus alternativas

Entendido. Vamos a desarrollar la sección sobre el dominio del problema y los tipos de herencia, incluyendo las alternativas y cómo se gestiona en diferentes lenguajes.

El Dominio del Problema

Antes de sumergirnos en las complejidades de la herencia y otras relaciones, es crucial comprender el **dominio del problema**. El dominio del problema se refiere al área de conocimiento o la realidad

para la que estamos construyendo nuestro sistema. Modelar el dominio del problema implica entender las entidades, conceptos, reglas de negocio y procesos relevantes de ese mundo real. El dominio del problema lo solemos acotar en el documento de Especificación de Requerimientos del Sistema (ERS, IEEE 830).

Una de las primeras cosas que tenemos que tener claras a la hora de modelar un dominio de problema es identificar cómo las entidades se relacionan entre sí, especialmente en términos de generalización y especialización. Aquí es donde entra en juego el concepto de herencia, pero no es la única herramienta.

Al modelar, buscamos capturar las características y comportamientos esenciales de estas entidades, así como las interacciones entre ellas. Esto se traduce en la creación de **clases** que representan estas entidades, con sus **atributos** y **métodos**, y el establecimiento de **relaciones** entre ellas. Una vez que hemos identificado estas relaciones, podemos empezar a considerar cómo se expresarán en nuestro diseño.

Tipos de Herencia

La **herencia** es un mecanismo fundamental de la POO que permite a una clase (subclase o clase hija) adquirir las propiedades (atributos) y comportamientos (métodos) de otra clase (superclase o clase padre). Esto modela una relación "es un tipo de" (por ejemplo, un `Perro` *es un tipo de* `Animal`). Sin embargo, la herencia no siempre es sencilla y presenta varias formas y consideraciones importantes.

Herencia Múltiple vs. Herencia Simple más Interfaces

Esta es una de las primeras y más importantes decisiones de diseño cuando se considera la herencia:

- **Herencia Simple:** Una clase solo puede heredar de **una única** superclase. Este enfoque es el más común en muchos lenguajes orientados a objetos (Java, C#, Smalltalk).
  - **Ventajas:** Simplifica la jerarquía de clases, reduce la complejidad del "diamante de la muerte" (problema de ambigüedad cuando una clase hereda el mismo método de dos ancestros distintos que a su vez heredan de un mismo ancestro común), y es más fácil de mantener.
  - **Desventajas:** Puede llevar a la duplicación de código si una clase necesita funcionalidades de múltiples fuentes.
- **Herencia Múltiple:** Una clase puede heredar de **múltiples** superclases, combinando sus características y comportamientos. Lenguajes como C++ y Python la soportan.

    - **Ventajas:** Permite reutilizar código de múltiples jerarquías, lo que puede resultar en un diseño más conciso.
    - **Desventajas:** Aumenta la complejidad, especialmente con el problema del "diamante de la muerte", donde puede haber ambigüedad sobre qué implementación de un método heredar si varias superclases lo tienen. Esto puede llevar a un código más difícil de entender y depurar.
- **Herencia Simple más Interfaces:** Muchos lenguajes (como Java y C#) optan por la herencia simple pero compensan sus limitaciones con el uso de **interfaces**. Una interfaz define un contrato de métodos que una clase debe implementar, sin proporcionar una implementación concreta. Una clase puede implementar **múltiples interfaces**.
    - **Ventajas:** Permite a una clase adoptar múltiples "roles" o comportamientos sin los problemas de ambigüedad de la herencia múltiple de clases. Promueve el polimorfismo y un diseño más flexible. Es una forma de lograr "herencia de comportamiento" sin "herencia de estado".
    - **Desventajas:** Las interfaces no pueden proporcionar implementaciones por defecto de atributos o métodos (aunque esto ha evolucionado con métodos por defecto en Java 8+ o propiedades en C#), lo que a veces requiere más código.

> **Actividad**
> Busca 3 ejemplos donde tenga sentido la herencia múltiple. Diseña una alternativa para cada uno de estos ejemplos usando Herencia simple más interfaces.

Herencia Privada o Herencia Pública (En C++)

Este concepto es específico de lenguajes como C++ y define cómo los miembros (atributos y métodos) de la superclase son accesibles desde la subclase y desde fuera de ella:

- **Herencia Pública (`public`):** Es el tipo de herencia más común y representa la relación "es un tipo de". Los miembros públicos de la superclase permanecen públicos en la subclase, y los miembros protegidos permanecen protegidos. Permite el polimorfismo y la sustitución de Liskov.
- **Herencia Protegida (`protected`):** Los miembros públicos y protegidos de la superclase se convierten en miembros protegidos en la subclase. Esto significa que los miembros heredados son accesibles dentro de la subclase y por sus propias subclases, pero no desde fuera de la jerarquía.
- **Herencia Privada (`private`):** Los miembros públicos y protegidos de la superclase se convierten en miembros privados en la subclase. Esto significa que la subclase puede usar esos miembros internamente, pero no son accesibles desde fuera de la subclase, ni siquiera por las subclases de esta. La herencia privada modela una relación "implementado en términos de" o "contiene una" más que "es un tipo de", funcionando casi como una composición interna.

Herencia frente a Composición

Esta es una de las decisiones de diseño más importantes y a menudo debatidas en la Programación Orientada a Objetos: ¿cuándo usar herencia y cuándo usar composición?

- **Herencia:** Modela una relación **"es un tipo de" (is-a)**. Una `Motocicleta` *es un tipo de* `Vehículo`. Implica una fuerte acoplamiento entre la superclase y la subclase, ya que la subclase hereda tanto la interfaz como la implementación de la superclase.
  - **Cuándo usarla:** Cuando la subclase es verdaderamente una especialización de la superclase y quieres reutilizar su implementación, a menudo junto con el polimorfismo.
- **Composición:** Modela una relación **"tiene un" (has-a)**. Una `Motocicleta` *tiene un* `Motor`. Implica que una clase contiene una instancia de otra clase (o varias) como uno de sus atributos. La funcionalidad se delega al objeto compuesto.
  - **Cuándo usarla:** Cuando la relación es de "todo-parte" o cuando una clase necesita la funcionalidad de otra pero no es un tipo de ella. Promueve un **acoplamiento bajo** y una mayor **flexibilidad**, ya que la implementación de la parte puede cambiarse sin afectar al todo (siempre que la interfaz de la parte se mantenga).
  - **Principio "Prefiere Composición sobre Herencia":** Es una regla de oro en el diseño de software. La composición suele ser más flexible y menos propensa a problemas de mantenimiento que la herencia profunda, ya que permite cambiar el comportamiento en tiempo de ejecución y reduce la fragilidad de las jerarquías.

Puedes ver el vídeo de CodeAesthetic sobre este tema en este enlace.

> **Actividad**
> Convierte los tres ejemplos de la actividad anterior sobre herencias en sistemas que usen la composición.

> **Actividad**
> Busca qué es, en el contexto de la programación, el acoplamiento. Razona qué tiene que ver con la herencia y la composición. Esta información se puede extraer del vídeo anterior.

Lenguajes Orientados a Objetos sin Herencia de Clases (como Rust)

Algunos lenguajes modernos, si bien son orientados a objetos en el sentido de encapsulación de datos y comportamiento, han optado por **no incluir la herencia de clases tradicional** para evitar sus problemas de complejidad y fragilidad.

- **Rust:** Se centra en la seguridad de la memoria y la concurrencia. No tiene herencia de clases. En su lugar, fomenta el uso de:

- **`Structs` (estructuras):** Para definir la estructura de los datos (atributos).
- **`Impl` (implementaciones):** Para definir métodos asociados a los `structs`.
- **`Traits` (rasgos):** Son similares a las interfaces o mixins. Un `trait` define un conjunto de métodos que un tipo debe implementar. Permiten el polimorfismo (un objeto puede ser tratado como un `trait` si implementa ese `trait`) y la reutilización de comportamiento a través de implementaciones por defecto en `traits` o al "mezclar" `traits` en `structs`. Esto es una forma de **polimorfismo de inclusión** sin la rigidez de la herencia de clases.

Lenguajes Orientados a Objetos usando Prototipos (como JavaScript)

A diferencia de los lenguajes basados en clases, que usan "planos" para crear objetos, los lenguajes basados en prototipos crean objetos directamente a partir de otros objetos existentes.

- **JavaScript:** Históricamente, JavaScript se basó en **prototipos** para la herencia. Cada objeto tiene una propiedad interna `[[Prototype]]` (o `__proto__`) que apunta a otro objeto. Cuando se accede a una propiedad o método de un objeto y no se encuentra en el propio objeto, JavaScript busca en su prototipo, y luego en el prototipo del prototipo, y así sucesivamente, formando una **cadena de prototipos**.
  - **Creación de objetos:** Se pueden crear objetos vacíos y añadirles propiedades, o usar la sintaxis `Object.create(otroObjeto)` para crear un objeto cuyo prototipo sea `otroObjeto`.
  - **"Herencia" basada en prototipos:** Un objeto "hereda" propiedades y métodos de su prototipo. No hay una distinción clara entre "clase" e "instancia" como en los lenguajes basados en clases.
  - **Sintaxis `class` en JavaScript (ES6+):** Aunque JavaScript introdujo la palabra clave `class` en ECMAScript 2015 (ES6), esto es principalmente **azúcar sintáctico** (syntactic sugar) sobre el modelo de herencia basado en prototipos existente. Por debajo, sigue funcionando con prototipos; la sintaxis `class` simplemente proporciona una forma más familiar para los programadores de lenguajes basados en clases de definir constructores y métodos.

Modelar la estructura de clases de un sistema requiere una comprensión profunda de estas opciones y sus implicaciones. La elección entre herencia, composición, y los diferentes mecanismos que ofrecen los lenguajes es crucial para construir sistemas robustos, flexibles y mantenibles.

Técnicas de Modelado con Diagramas de Clases UML

Modelar un sistema con diagramas de clases UML no es solo dibujar cajas y flechas; es un proceso iterativo y reflexivo para entender y comunicar la estructura de un software. Implica identificar las

entidades, sus características y cómo interactúan.

1. Identificación de Clases Candidatas

El primer paso es reconocer las posibles clases en el sistema. Esto se puede hacer de varias maneras:

- **Análisis de Sustantivos:** Lee la descripción del problema o los requisitos del sistema y subraya todos los **sustantivos** (personas, lugares, cosas, conceptos, eventos). Cada sustantivo es un candidato potencial para una clase.
  - **Ejemplo:** En un "sistema de gestión de biblioteca", sustantivos como "libro", "miembro", "préstamo", "autor", "editorial" son posibles clases.
- **Análisis de Tarjetas CRC (Class, Responsibility, Collaboration):** Una técnica sencilla y colaborativa. Cada tarjeta representa una clase.
  - **Clase (C):** Nombre de la clase.
  - **Responsabilidad (R):** Qué sabe la clase y qué hace (sus atributos y métodos).
  - **Colaboración (C):** Con qué otras clases interactúa para cumplir sus responsabilidades.
  - Es útil para un brainstorming rápido y para asegurar que cada clase tiene una **responsabilidad única** (relacionado con el Principio de Responsabilidad Única de SOLID).
- **Identificación de Roles:** Las personas o sistemas externos que interactúan con tu sistema pueden ser clases (ej., `Administrador`, `Cliente`).
- **Identificación de Eventos y Transacciones:** Acciones significativas que ocurren en el sistema (ej., `Venta`, `Devolucion`, `Registro`).

2. Definición de Atributos y Comportamientos (Métodos)

Una vez que se tienen las clases candidatas, el siguiente paso es dotarlas de contenido:

- **Atributos:** Piensa en las **propiedades** que necesita conocer cada instancia de la clase para ser completamente descrita.
  - ¿Qué datos necesita almacenar este objeto?
  - ¿Qué características lo distinguen?
  - **Reflexión sobre el tipo de dato y visibilidad:** Decide el tipo de dato apropiado (string, int, Date, etc.) y la visibilidad (`+` público, `-` privado, `#` protegido). La mayoría de los atributos deberían ser **privados** para asegurar la **encapsulación**.
- **Métodos:** Identifica las **acciones** que la clase puede realizar o que se pueden realizar sobre ella.
  - ¿Qué puede hacer este objeto?
  - ¿Qué operaciones se necesitan para manipular sus atributos o interactuar con otras clases?

- **Reflexión sobre el comportamiento y visibilidad:** Decide los parámetros que necesita el método, el tipo de retorno y su visibilidad. Los métodos que forman la API pública de la clase suelen ser **públicos**.

3. Modelado de Relaciones entre Clases

Las clases no existen en el vacío; interactúan. Modelar estas interacciones es crucial:

- **Asociaciones:** La relación más genérica entre dos clases. Indica que los objetos de una clase están conectados o utilizan objetos de otra.
  - **Identificación:** Busca verbos que conecten sustantivos (ej., "un cliente *realiza* un pedido").
  - **Cardinalidad:** Es fundamental especificar cuántas instancias de una clase están relacionadas con cuántas instancias de otra (1 a 1, 1 a muchos, muchos a muchos).
  - **Navegabilidad:** Indica la dirección en la que se puede acceder desde un objeto a otro. Por defecto, es bidireccional, pero a menudo es unidireccional en la implementación.
- **Agregación (o--):** Representa una relación "todo-parte" donde las partes pueden existir independientemente del todo.
  - **Identificación:** "Un coche *tiene* ruedas" (las ruedas existen aunque el coche se desguace).
- **Composición (*--):** Una forma fuerte de agregación donde las partes no pueden existir sin el todo. Si el todo es destruido, las partes también lo son.
  - **Identificación:** "Una casa *tiene* habitaciones" (las habitaciones no existen si la casa se derrumba).
- **Dependencia (..>):** La relación más débil. Una clase "usa" otra, generalmente porque un método de una clase recibe un objeto de la otra como parámetro, o lo crea localmente. No implica que un objeto sea parte de otro.
  - **Identificación:** "Un cliente *envía* un email" (la clase `Cliente` usa la clase `EmailSender` para su método `enviarEmail`).

4. Refinamiento y Verificación

El modelado es un proceso iterativo. Rara vez se acierta a la primera. Por ello, es importante ir refinándolas poco a poco.

- **Aplicación de Principios de Diseño (SOLID):**
  - **SRP (Responsabilidad Única):** ¿Cada clase tiene una única razón para cambiar? Si una clase tiene demasiados atributos o métodos no relacionados, considera dividirla.
  - **OCP (Abierto/Cerrado):** ¿Pueden mis clases extenderse sin modificación? Las interfaces y clases abstractas son clave aquí.

    - **LSP (Sustitución de Liskov):** (Aunque aún no hemos visto herencia, es bueno tenerlo en mente para el futuro).
    - **ISP (Segregación de Interfaces):** ¿Tus interfaces son lo suficientemente pequeñas y específicas para evitar forzar a los clientes a implementar métodos que no necesitan?
    - **DIP (Inversión de Dependencias):** ¿Las clases de alto nivel dependen de abstracciones en lugar de detalles concretos?
- **Revisión con Escenarios de Uso (Casos de Uso):** Recorre los casos de uso principales del sistema y simula cómo los objetos interactuarían en tu diagrama de clases.
    - ¿Se pueden realizar todas las acciones?
    - ¿Hay clases que tienen demasiadas responsabilidades?
    - ¿Faltan atributos o métodos?
    - ¿Las relaciones son lógicas y tienen la cardinalidad correcta?
- **Coherencia y Claridad:** Asegúrate de que el diagrama sea fácil de entender. Usa nombres claros para clases, atributos y métodos. Evita complejidades innecesarias.
- **Iteración:** El modelado es un ciclo: identifica, define, relaciona, revisa y repite.

> **Actividad**
> Modela un diagrama de clases a partir del siguiente enunciado.
>
> *Sistema de Gestión de Alquiler de Coches*
>
> Se te ha encargado desarrollar un sistema de gestión para una empresa de alquiler de coches. La empresa tiene un nombre y varias sucursales distribuidas en distintas ciudades. Cada sucursal está identificada por su ciudad y dirección.
>
> Cada sucursal cuenta con una flota de vehículos de diferentes tipos (por ejemplo, sedán, SUV, furgoneta). Cada vehículo está identificado por una matrícula, modelo, marca, año de fabricación, y el número de asientos. Además, cada coche puede estar disponible o alquilado.
>
> El sistema debe permitir:
>
> - Registrar la información de nuevos vehículos en una sucursal.
> - Alquilar un coche a un cliente. Para ello, se debe registrar el nombre del cliente, su identificación y el coche que ha alquilado.
> - Devolver coches alquilados.
> - Consultar la disponibilidad de vehículos en cada sucursal.
> - Filtrar los vehículos por tipo, marca o número de asientos.
>
> Además, el sistema debe llevar un control de la cantidad de vehículos disponibles y en uso por cada sucursal, y debe permitir acceder a la información de cada coche, sucursal y cliente.

De Diagrama Entidad-Relación a Diagrama de Clases UML: Una Aproximación

La conversión de un **Diagrama Entidad-Relación (ER)** a un **Diagrama de Clases UML** es un paso común en el diseño de software, especialmente cuando se parte de un modelo de datos conceptual o lógico para desarrollar una aplicación orientada a objetos. Aunque no es una traducción 1:1 perfecta (ya que los Diagramas ER se centran en los datos y las bases de datos, mientras que los diagramas de clases modelan el comportamiento y la estructura del código), hay una serie de reglas de aproximación muy útiles.

El objetivo es transformar las entidades y relaciones del Diagrama ER en clases, atributos y asociaciones en el diagrama de clases, preparándonos para la implementación en un lenguaje orientado a objetos.

Reglas de Conversión Aproximadas

A continuación, se describen las correspondencias más habituales:

1. Entidades -> Clases

- **Cada entidad fuerte (o entidad normal) en el Diagramas ER se convierte en una clase en el diagrama de clases UML.**
- El **nombre de la entidad** se convierte en el nombre de la clase. Se recomienda usar **UpperCamelCase** (PascalCase) para los nombres de las clases.
  Ejemplo:
  - Entidad: `CLIENTE` -> Clase: `Cliente`
  - Entidad: `PRODUCTO` -> Clase: `Producto`

▶ **Haz click aquí para ver el código plantuml**

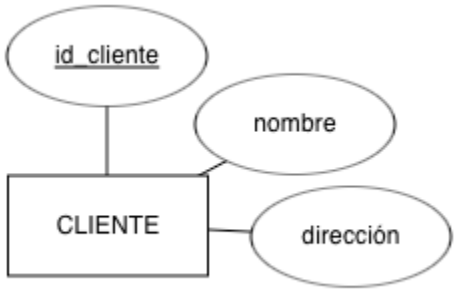
2. Atributos de Entidades -> Atributos de Clases

- **Cada atributo de una entidad se convierte en un atributo de la clase correspondiente.**
- El **nombre del atributo** se mantiene, pero se recomienda usar **lowerCamelCase**.
- El **tipo de dato** del atributo (ej., `VARCHAR(50)`, `INT`, `DATE`) se traduce al tipo de dato más apropiado en el lenguaje de programación objetivo (ej., `string`, `int` / `number`, `Date`).

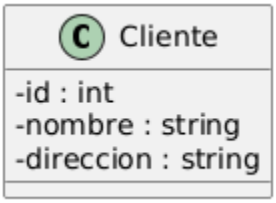
- Los **atributos clave primaria (PK)** se suelen representar como atributos normales en la clase, a menudo con un estereotipo `<<PK>>` o simplemente identificados por contexto. Su unicidad y la imposibilidad de ser nulos se gestionarán a nivel de implementación (constructores, lógica de negocio).
- Los **atributos clave foránea (FK)** que solo sirven para establecer una relación (es decir, no son datos intrínsecos de la entidad) **no se representan directamente como atributos** en la clase receptora en UML. En su lugar, la relación se modela como una **asociación** entre las clases. Si la FK representa un valor que es significativo para el objeto más allá de la relación (ej., `idPaisNacimiento` en una persona, cuando el `Pais` es otra clase), entonces sí podría mantenerse como atributo.

  Ejemplo:

Este diagrama ER:



Se convierte en este diagrama de clases UML:



▶ **Haz click aquí para ver el código plantuml**

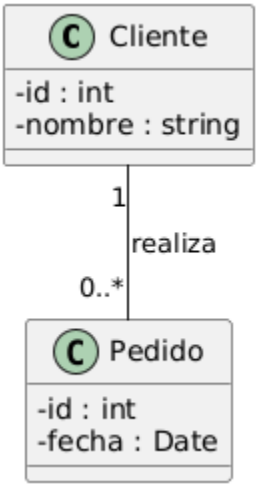
3. Relaciones -> Asociaciones

- **Cada relación entre entidades en el Diagramas ER se convierte en una asociación entre las clases correspondientes en UML.**
- La **cardinalidad** de la relación en el Diagramas ER se traduce directamente a la cardinalidad de la asociación en UML.
  - `1:1` -> `1..1` o `1` en ambos extremos
  - `1:N` -> `1` en el lado "uno", `*` (o `0..*` o `1..*`) en el lado "muchos"
  - `N:M` -> `*` en ambos extremos

- **Nombres de Roles:** Si la relación en el Diagramas ER tiene roles, estos se pueden usar como nombres de rol en los extremos de la asociación en UML.

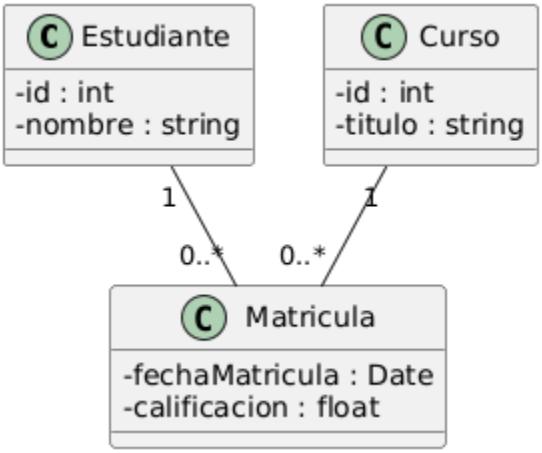
Ejemplos:

- **1:N (Uno a Muchos):** Un `Cliente` **realiza** `Pedidos`.



▶ **Haz click aquí para ver el código plantuml**

- **N:M (Muchos a Muchos):** Un `Estudiante` **se matricula en** `Cursos`.
  - En un Diagrama ER, una relación N:M a menudo se resuelve con una tabla intermedia. En UML, esto se modela como una **clase de asociación**.



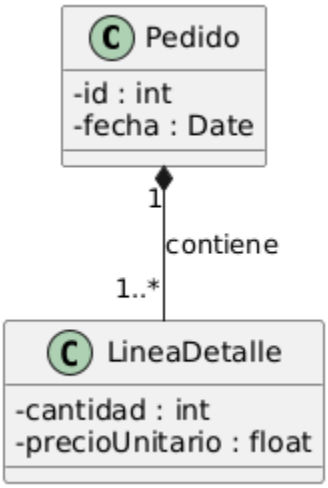
▶ **Haz click aquí para ver el código plantuml**

Aquí, `Matricula` es una clase que contiene atributos propios de la relación (fecha, calificación) y se asocia tanto con `Estudiante` como con `Curso`.

4. Entidades Débiles -> Clases (con Composición/Agregación)

- Las **entidades débiles** (que dependen de la existencia de otra entidad fuerte) a menudo se modelan en UML utilizando **composición** (rombo relleno) o **agregación** (rombo vacío), dependiendo de si la entidad débil puede o no existir independientemente de la entidad fuerte "propietaria".
- La **composición** es más común para entidades débiles, ya que su existencia está ligada a la del propietario.

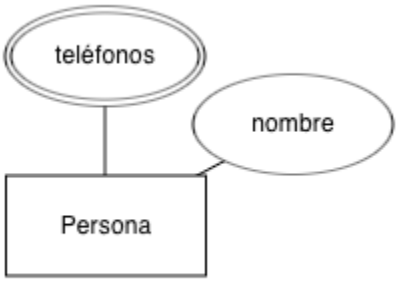
  Ejemplo: Un `Pedido` tiene `LineasDeDetalle`. Si el pedido se elimina, sus líneas de detalle también.



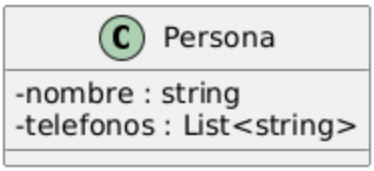
▶ **Haz click aquí para ver el código plantuml**

5. Atributos Multivaluados -> Colecciones o Clases Separadas

- Si un atributo en el Diagrama ER puede tener múltiples valores (ej., `telefono` de una persona, que puede tener varios), en UML se modela como una **colección** (ej., `List<string>`, `Set<string>`) del atributo dentro de la clase, o, si los valores tienen atributos propios (ej., un teléfono con `tipo` y `numero`), como una **clase separada** relacionada por asociación o composición.

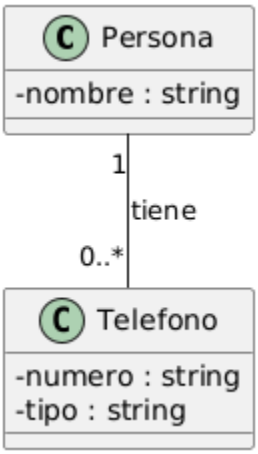


Ejemplo (como colección):



▶ **Haz click aquí para ver el código plantuml**

Ejemplo (como clase separada si `Telefono` tiene más propiedades):



▶ **Haz click aquí para ver el código plantuml**

Consideraciones Adicionales

- **Métodos:** Los Diagramas ER se centran en datos, por lo que no representan comportamientos. Al convertir a UML, deberás **añadir los métodos** a las clases basándote en los requisitos funcionales del sistema (qué acciones se realizan con esos datos).
- **Encapsulación:** En UML, se debe definir la visibilidad (`+` , `-` , `#` , `~`) para atributos y métodos, priorizando el encapsulamiento (atributos privados, métodos públicos para interactuar).
- **Refinamiento:** La primera aproximación es solo eso, una aproximación. El diagrama de clases resultante debe ser **refinado** para incorporar principios de diseño de POO (como SOLID), patrones de diseño y las especificidades del lenguaje de programación y el dominio del problema.

Puedes usar ERDPlus para crear diagramas entidad relación fácilmente.

> **Actividad**
> Convierte el enunciado del ejercicio anterior en un Diagrama Entidad Relación en la medida de lo posible.

> **Actividad**
> Dados los siguientes Diagramas Entidad Relación, modela los diagramas de clases

> correspondiente. Añade los métodos que creas oportunos según la semántica de la base de datos (por ejemplo, un pájaro "vuela", aunque en el diagrama Entidad Relación eso no se represente).

Diagrama 1

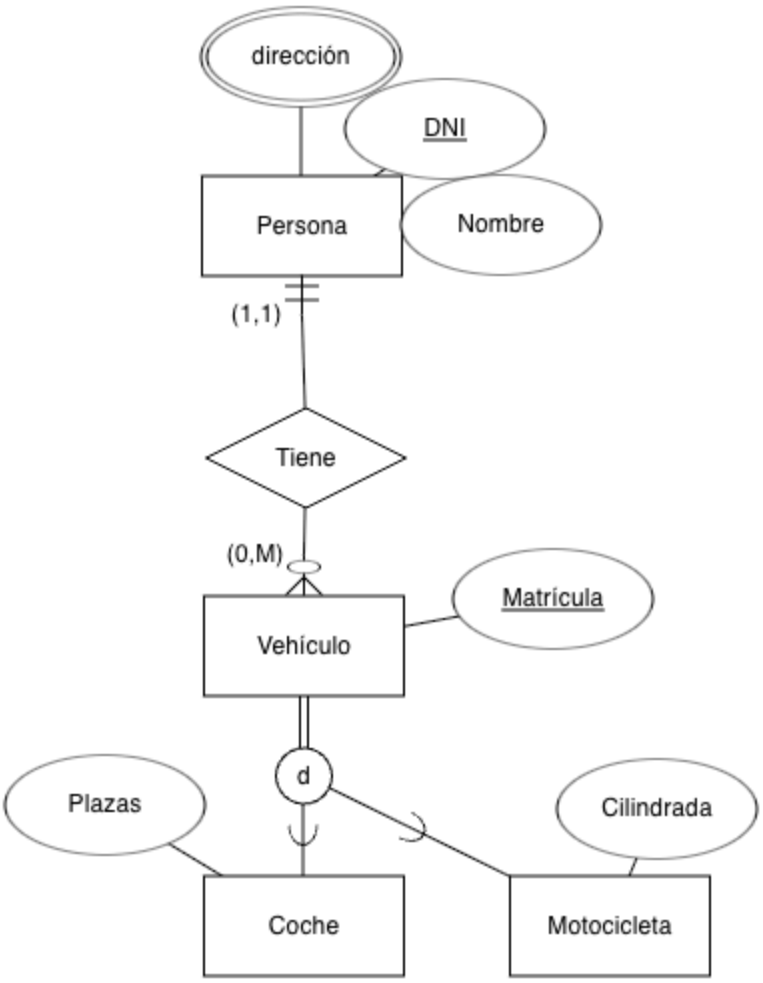
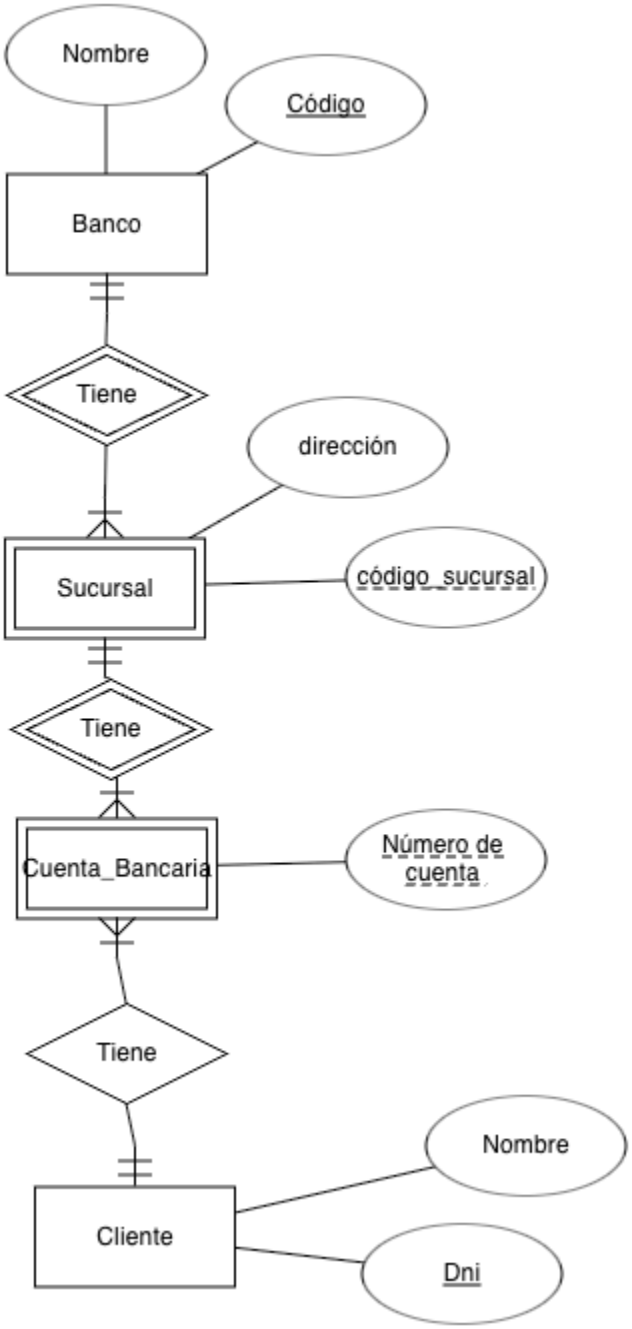


Diagrama 2


Nombre
Código
Banco
Tiene
dirección
Sucursal
código_sucursal
Tiene
Cuenta_Bancaria
Número de cuenta
Tiene
Nombre
Cliente
Dni

Reto cooperativo de la Unidad de Programación 03

Análisis de sistemas y diseño orientado a objetos

Formación del equipo

Para esta unidad de programación, la formación de los equipos se hará a través del método *Dinámica de los colores*. Observa la tabla:

¿Con qué colores te identificas más? Escoge en la encuesta en FORMS tu primera y tu segunda opción. En base a ella, se crearán los equipos de la clase intentando tener representados todos los colores.

Tarea

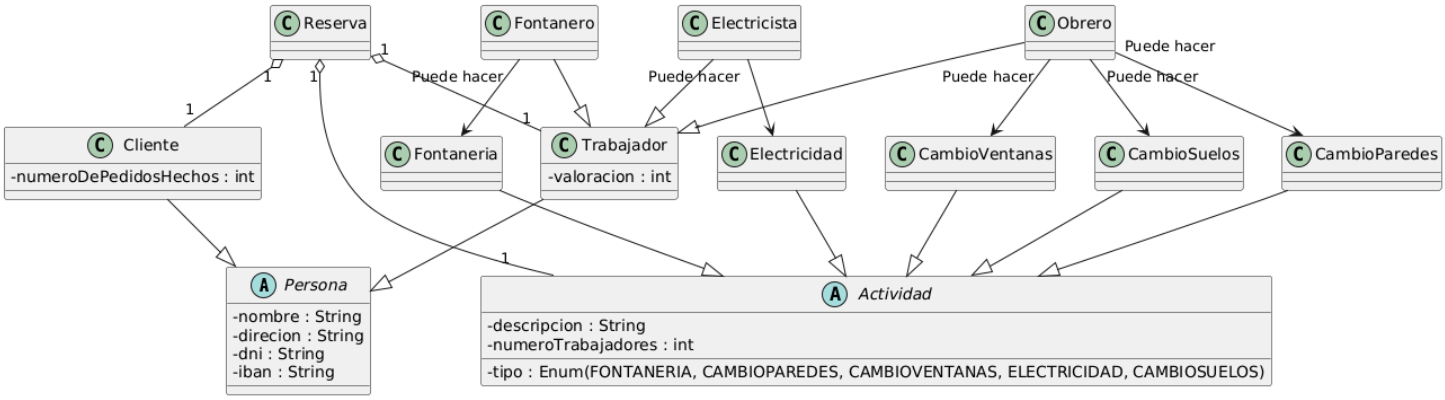
Continúa el proyecto que se te ha asignado. Este proyecto fue comenzado por otro equipo, que se encargó del **Diseño de los diagramas de comportamiento**. Tu labor es:

- Corregir posibles errores del producto entregado por el equipo anterior antendiendo a la especificación dada
- Realizar el diseño de las clases UML
- Proponer una traducción del diagrama de clases a un diagrama Entidad Relación

- Terminar el documento ERS IEEE830

Reto individual

Explica detalladamente en qué consiste el sistema representado mediante el siguiente diagrama de clases



▶ **Haz click aquí para ver el código plantuml**

Crea un diagrama de clases acorde a la siguiente especificación

Una empresa de reparaciones informáticas ha decidido modernizar su gestión y presencia online mediante el desarrollo de una nueva página web. Esta plataforma será fundamental para optimizar sus operaciones y la interacción con sus clientes.

La empresa ofrece una amplia gama de servicios especializados, y es crucial que cada uno de ellos pueda ser identificado rápidamente mediante un **código único**. Entre los servicios que proporcionan se incluyen la recuperación de datos de discos duros dañados, el análisis de averías, reparaciones que se realizan "en el momento", la intermediación en reparaciones más complejas, asesoramiento para el montaje de ordenadores, y reparaciones tanto de dispositivos móviles como de videoconsolas.

En cuanto a su plantilla, la empresa cuenta con diversos tipos de **trabajadores**, cada uno de ellos identificado de forma unívoca por su **número de DNI**. De todos ellos se necesita registrar su información personal relevante, como sus **datos postales**, sus **datos bancarios** y su **salario**. Dentro de la organización, se distinguen principalmente dos roles: los **trabajadores gestores**, cuya función

es la administración y coordinación general de los servicios de reparación, y los **trabajadores reparadores**, que son quienes llevan a cabo las reparaciones directamente en el taller. Es importante destacar que, entre los trabajadores reparadores, existen dos especializaciones claras: algunos son **expertos en ordenadores**, mientras que otros están específicamente capacitados como **expertos en móviles**.

La página web debe facilitar varias funcionalidades clave para el día a día de la empresa. Los trabajadores gestores necesitarán una herramienta que les permita **asignar los diferentes servicios de reparación** a los trabajadores reparadores más adecuados. Por su parte, los trabajadores reparadores deberán poder **consultar las reparaciones que tienen asignadas** y, fundamentalmente, **actualizar el estado** de cada una de ellas a medida que avanzan en el proceso. La interacción con los **clientes de la tienda** también es un pilar fundamental del nuevo sistema; de ellos se almacenará su **correo electrónico**, sus **datos postales** y sus **datos bancarios**, y se les debe proporcionar un medio para que puedan ponerse en contacto con la empresa a través de la web. Los correos recibidos por esta vía serán respondidos por los trabajadores gestores.

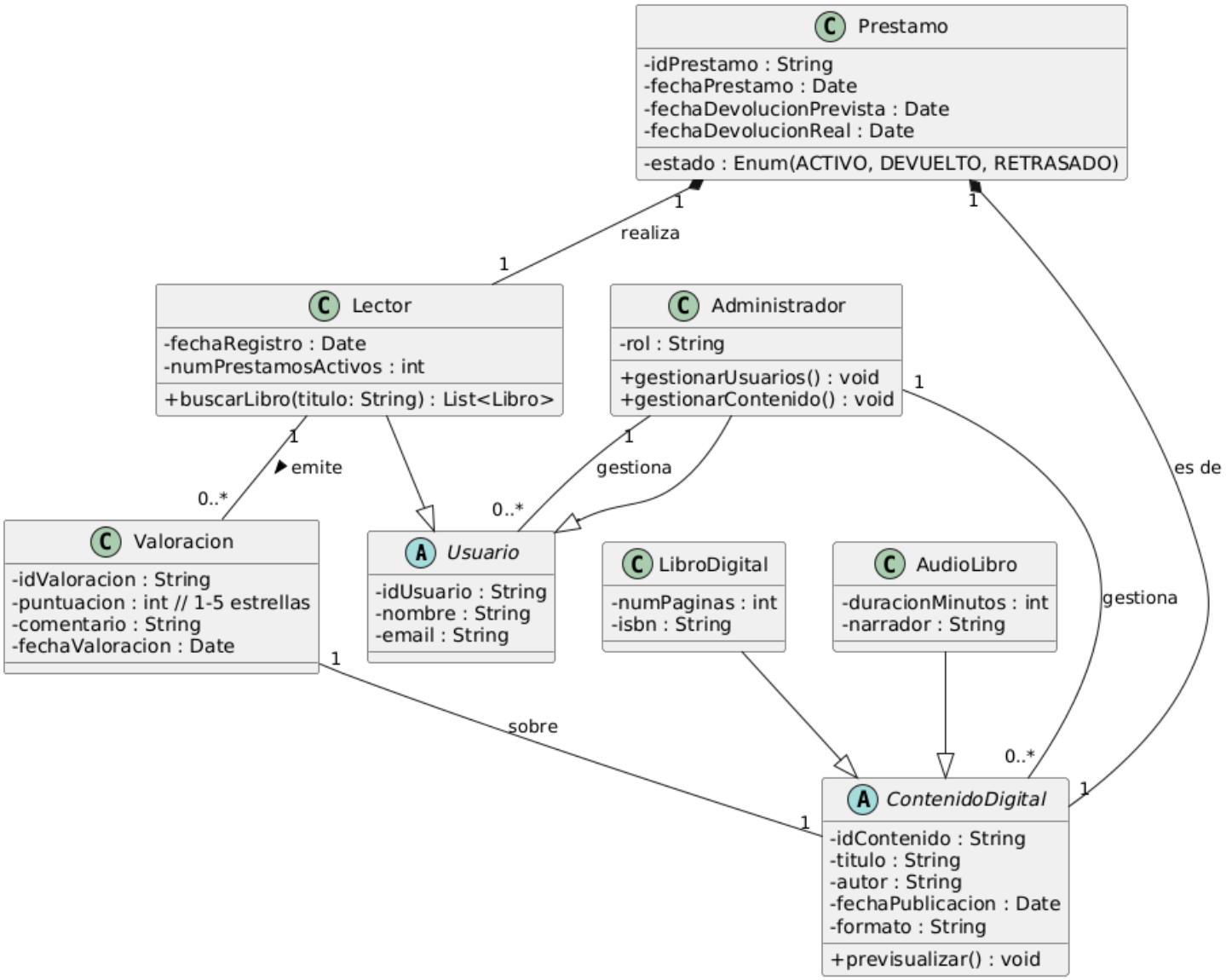
Finalmente, el sistema debe contemplar la creación de un **usuario especial**. Este usuario tendrá **acceso total a todas las acciones** que pueden realizar los trabajadores, lo que le permitirá una supervisión completa del sistema. Adicionalmente, y con un propósito de depuración y verificación, este usuario especial podrá **simular la interacción de un cliente de la tienda** para comprobar el correcto funcionamiento de las funcionalidades orientadas al usuario final.

Rubrica

Ejercicio	ÍTEM	Criterio Evaluación	PESO
1	Identifica los componentes de las clases	5a	1
1	Identifica las relaciones	5a	1
1	Calidad en la explicación	5c	1
2	Adecuación al problema	5c	1
2	Sintaxis adecuada	5b	1
2	Formación correcta de las clases	5b,5d	2
2	Formación correcta de las relaciones	5b,5d	2
2	Calidad de la presentación	5b	1

Examen

EJERCICIO 1: Explica detalladamente en qué consiste el sistema representado mediante el siguiente diagrama de clases



▶ **Haz click aquí para ver el código Plantuml**

EJERCICIO 2: Crea un diagrama de clases acorde a la siguiente especificación

Una cadena de gimnasios, "Fitness Global", planea un ambicioso proyecto para desarrollar una **aplicación móvil** que digitalice y mejore la experiencia de sus usuarios y la gestión interna. El objetivo es crear una herramienta integral que abarque desde la administración de membresías hasta la programación de clases y el seguimiento personalizado del progreso.

En primer lugar, la aplicación deberá gestionar a los **socios** del gimnasio. De cada socio se registrará su **nombre completo**, **fecha de nacimiento**, **dirección de correo electrónico** y un **número de teléfono** de contacto. Es fundamental controlar el estado de su **membresía**, incluyendo su **tipo** (por ejemplo, mensual, trimestral, anual) y su **fecha de vencimiento**, ya que esto determinará su acceso a las instalaciones y servicios. Además, cada socio tendrá la posibilidad de establecer **objetivos de entrenamiento** personalizados, como "perder peso", "ganar masa muscular" o "mejorar resistencia", los cuales podrán ser consultados por los entrenadores.

La oferta de "Fitness Global" se centra en **clases grupales** variadas, como yoga, spinning, zumba, etc. Cada clase se identificará por un **nombre único** y tendrá una **duración** específica. Es importante registrar el **horario** en que se imparte, el **instructor** que la dirige y la **capacidad máxima** de participantes. Los socios deberán poder **reservar su plaza** en las clases, y la aplicación debe asegurar que no se exceda la capacidad. Si una clase está llena, se deberá poder **apuntarse a una lista de espera**. Si una plaza queda libre, el primer socio de la lista de espera será notificado y se le asignará la plaza.

El equipo de "Fitness Global" está compuesto por **empleados** con diferentes roles. Todos los empleados se identificarán por un **código de empleado** interno y tendrán registrados su **nombre**, **apellidos**, **DNI** y **salario**. Dentro de la plantilla, se distinguen principalmente dos categorías: los **recepcionistas** y los **entrenadores**. Los recepcionistas serán los encargados de la **gestión de altas y bajas de socios**, así como de la **modificación de los datos de las membresías**. Los entrenadores, por su parte, tendrán la capacidad de **crear nuevas clases**, **modificar las existentes** y **consultar los objetivos de entrenamiento** de los socios para ofrecer un seguimiento más personalizado. Cada entrenador tendrá una o varias **especialidades** (por ejemplo, "yoga", "musculación", "pilates"), las cuales son relevantes para la asignación de clases.

La aplicación también debe permitir a los socios consultar su **historial de clases asistidas** y su **progreso** en relación con sus objetivos de entrenamiento (por ejemplo, un registro de su peso o de las repeticiones en ciertos ejercicios, si deciden introducirlo). Los recepcionistas deberán poder generar **informes sobre el número de membresías activas** y la **ocupación de las clases**.

Finalmente, se requiere un **administrador del sistema**. Este usuario especial tendrá **control total sobre todas las funcionalidades de la aplicación**, lo que significa que podrá realizar cualquier acción que puedan hacer tanto los recepcionistas como los entrenadores. Además, el administrador podrá **gestionar las cuentas de los empleados** (alta, baja y modificación de roles), y tendrá acceso a **estadísticas globales de uso** de la aplicación, como el número total de reservas o las clases más populares.

Rubrica

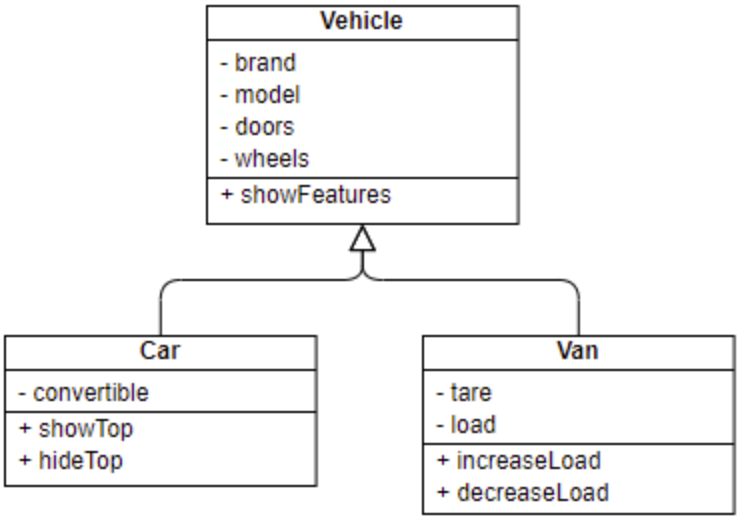
Ejercicio	ÍTEM	Criterio Evaluación	PESO
1	Identifica los componentes de las clases	5a	1
1	Identifica las relaciones	5a	1
1	Calidad en la explicación	5c	1
2	Adecuación al problema	5c	1
2	Sintaxis adecuada	5b	1
2	Formación correcta de las clases	5b,5d	2
2	Formación correcta de las relaciones	5b,5d	2
2	Calidad de la presentación	5b	1

Cuaderno de ejercicios 03

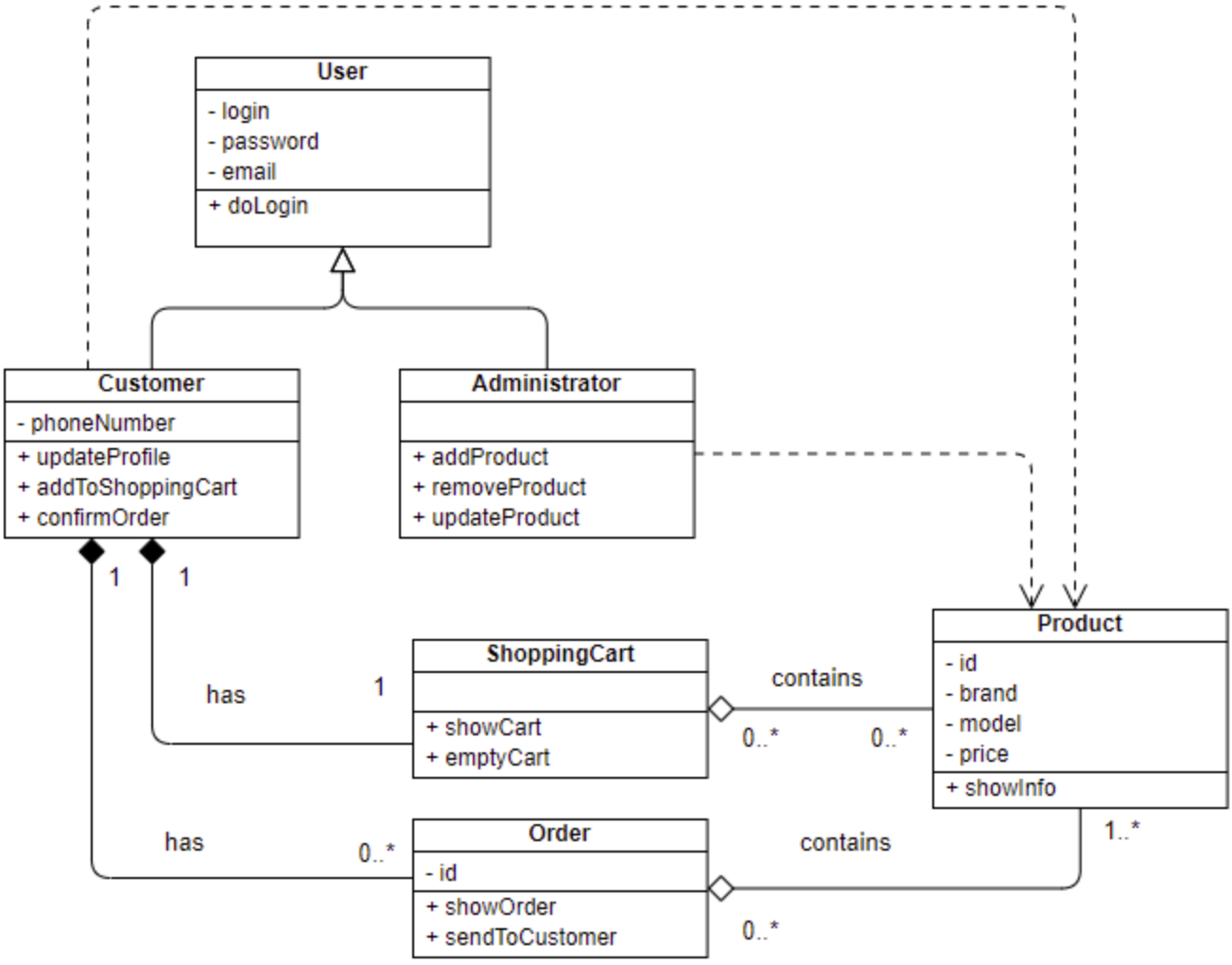
Ejercicios de diagramas

> **Actividad**
>
> 1. Explica los siguientes diagramas UML extraídos de diversas fuentes
> 2. Crea una versión equivalente (en la medida de lo posible) en Diagrama Entidad Relación

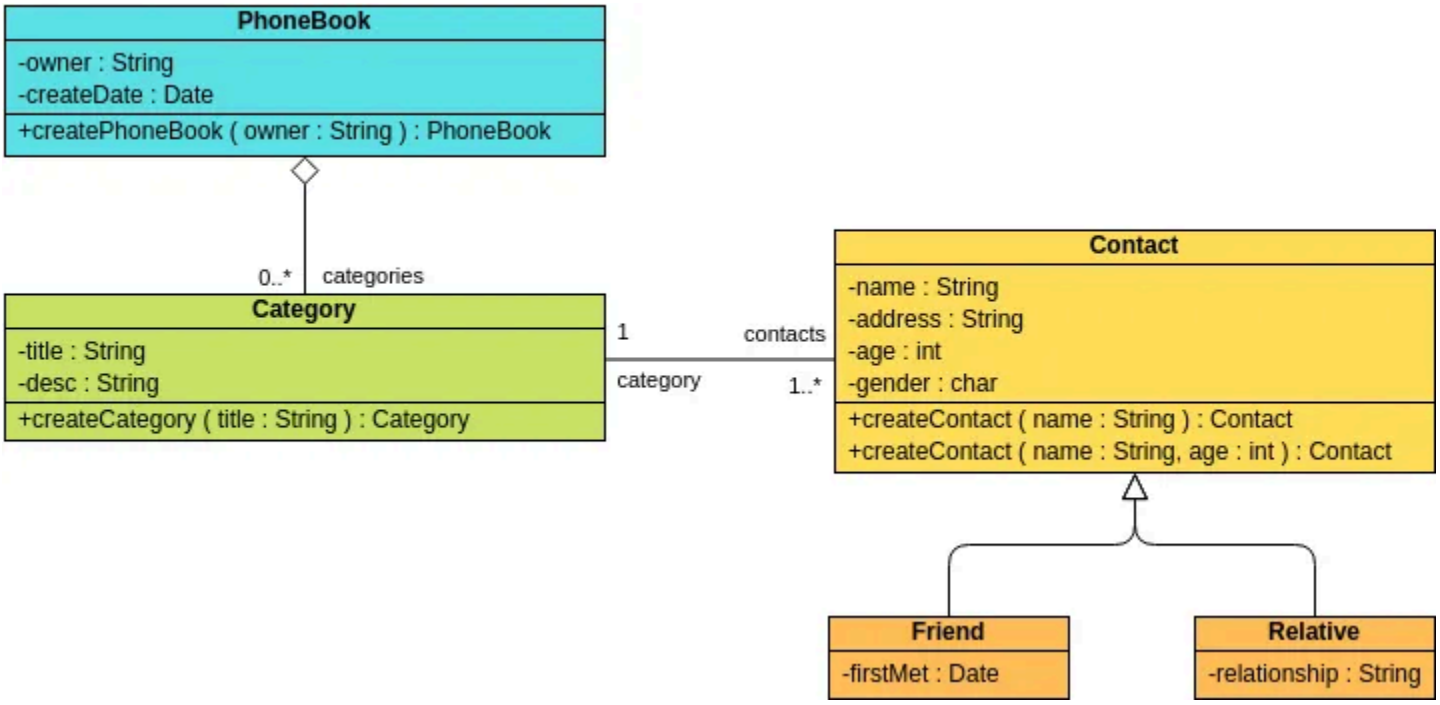
1



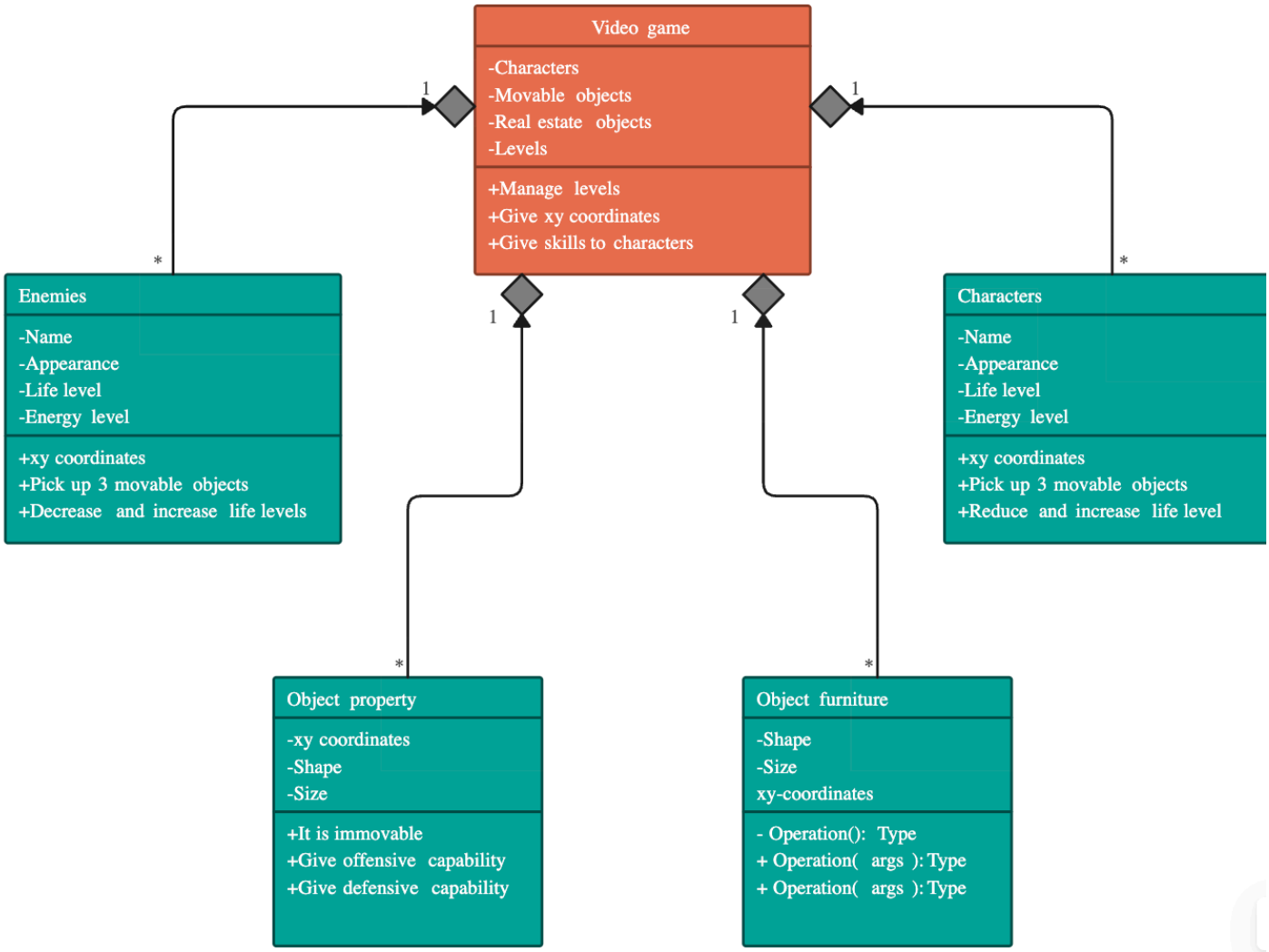
2


User
- login
- password
- email
+ doLogin
Customer
- phoneNumber
+ updateProfile
+ addToShoppingCart
+ confirmOrder
Administrator
+ addProduct
+ removeProduct
+ updateProduct
1
1
has
1
ShoppingCart
+ showCart
+ emptyCart
contains
0..*
0..*
Product
- id
- brand
- model
- price
+ showInfo
1..*
has
0..*
Order
- id
+ showOrder
+ sendToCustomer
contains
0..*

3


PhoneBook
-owner : String
-createDate : Date
+createPhoneBook (owner : String) : PhoneBook
0..*
categories
Category
-title : String
-desc : String
+createCategory (title : String) : Category
1
category
contacts
1..*
Contact
-name : String
-address : String
-age : int
-gender : char
+createContact (name : String) : Contact
+createContact (name : String, age : int) : Contact
Friend
-firstMet : Date
Relative
-relationship : String

4



Ejercicios de modelización

Ejercicio 1: Gestor de Liga de Fútbol

Se ha encargado desarrollar un gestor para una liga de fútbol. La liga debe estar identificada por un nombre y una temporada, definida por el año de inicio y el de finalización. La liga estará compuesta por un máximo de 22 equipos.

Cada equipo debe incluir información como su nombre, el número de partidos ganados, empatados y perdidos. A partir de estos datos, se podrá calcular la puntuación total de cada equipo según el sistema estándar de puntos (3 puntos por victoria, 1 por empate, 0 por derrota). Además, cada equipo debe tener entre 18 y 24 futbolistas.

Cada futbolista debe tener datos que lo identifiquen: nombre, nacionalidad, un número de identificación único, su posición en el campo (portero, defensa, centrocampista o delantero), el número de goles marcados y el número de partidos jugados.

Se requiere que solo exista una instancia de la liga (patrón singleton). Debes implementar la funcionalidad para añadir y eliminar equipos, así como para añadir y eliminar futbolistas de los equipos. Además, se debe poder acceder a la información completa de cada jugador, equipo o la liga misma.

El sistema debe proporcionar las siguientes funcionalidades:

- Mostrar los equipos en posiciones de descenso (los 4 últimos).
- Mostrar los equipos en posiciones de clasificación para competiciones europeas (los 4 primeros).
- Calcular los goles a favor de cada equipo.
- Identificar al máximo goleador ("pichichi") de la liga.

Ejercicio 2: Gestor de Aeropuertos

Se te ha pedido desarrollar un sistema para gestionar la operativa de un aeropuerto internacional. El aeropuerto debe estar identificado por un nombre, un código IATA de tres letras, y la ciudad donde se encuentra.

Cada aeropuerto gestiona varios vuelos. Un vuelo está identificado por un número único, la aerolínea operadora, el destino y el origen. También debe incluir la hora de salida, la hora de llegada estimada y el estado del vuelo (en hora, retrasado, cancelado).

Cada vuelo tiene una tripulación asignada, formada por al menos un piloto y dos auxiliares de vuelo. Los tripulantes deben incluir información como su nombre, nacionalidad, y su número de identificación único.

Se debe poder:

- Añadir y eliminar vuelos.
- Asignar o eliminar tripulantes de los vuelos.
- Consultar el estado de cualquier vuelo en cualquier momento.
- Filtrar los vuelos por su destino, origen o estado.
- Calcular el tiempo estimado de llegada para los vuelos en base a su hora de salida.

El sistema debe permitir acceder a la información completa de cada vuelo y cada tripulante.

Ejercicio 3: Sistema de Gestión de Turnos en un Hospital

Un hospital te ha solicitado desarrollar un sistema para gestionar los turnos de trabajo de su personal. El hospital está compuesto por diferentes departamentos (p. ej., Urgencias, Pediatría,

Cardiología), y cada uno tiene un nombre que lo identifica.

El personal del hospital incluye doctores, enfermeros y auxiliares. Cada empleado está identificado por su nombre, número de identificación, su cargo, y su especialidad (en el caso de los doctores). Cada empleado está asignado a un departamento específico.

El sistema debe gestionar los turnos de trabajo de cada empleado. Un turno tiene una fecha, una hora de inicio y una hora de fin. Cada empleado puede estar asignado a múltiples turnos, pero no puede tener más de un turno asignado en el mismo horario.

El sistema debe permitir:

- Asignar turnos a empleados y eliminarlos si es necesario.
- Consultar los turnos asignados a un empleado o a un departamento completo.
- Filtrar turnos por fecha o por departamento.
- Identificar empleados con exceso de horas de trabajo (más de 40 horas semanales).
- Consultar la disponibilidad de un empleado en un horario determinado.

Se debe acceder fácilmente a la información completa de cada empleado, sus turnos y los departamentos del hospital.

Ampliación: Diagramas de Objetos

Diagramas de Objetos

Los diagramas de objetos son una instantánea en el tiempo de tu sistema. A diferencia de los diagramas de clases, que muestran la estructura abstracta y las relaciones entre las clases, los **diagramas de objetos muestran las instancias concretas de esas clases (los objetos) en un momento específico de la ejecución**. Imagina que un diagrama de clases es como el plano de una casa, mientras que un diagrama de objetos es una fotografía de esa casa amueblada y con la gente dentro en un instante dado.

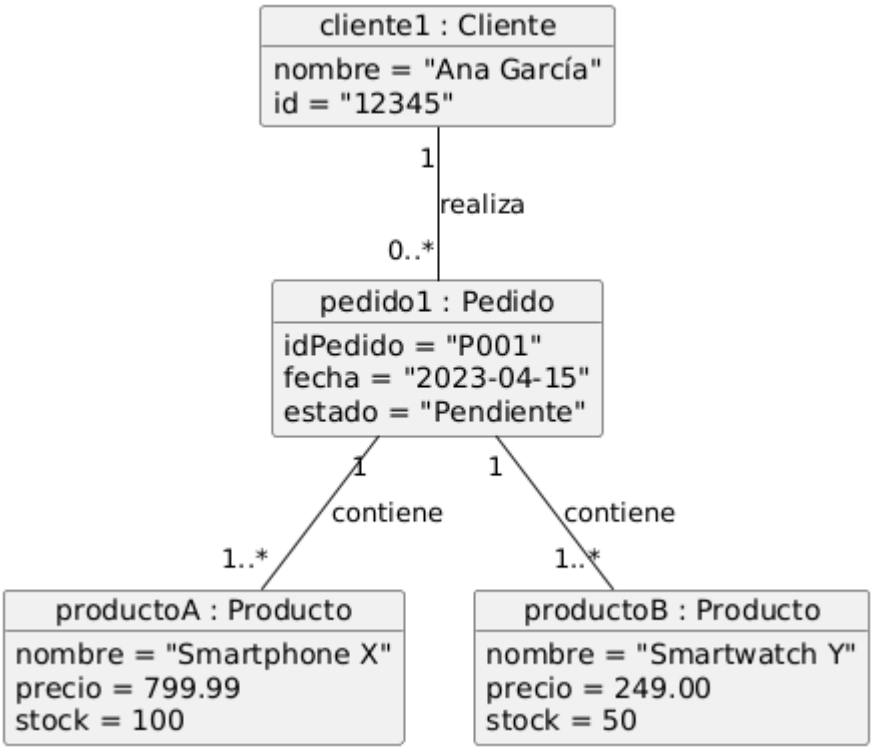
Estos diagramas son ideales para:

- **Ilustrar ejemplos concretos:** Muestran cómo un sistema se ve y se comporta con datos reales.
- **Validar el modelo de clases:** Ayudan a verificar si las clases y sus relaciones son capaces de representar los objetos y sus enlaces en escenarios específicos.
- **Comprender la configuración de un sistema:** Permiten visualizar el estado de los objetos y sus atributos en un momento determinado.

Estructura de un Diagrama de Objetos

Un diagrama de objetos se compone de:

- **Objetos:** Representados como rectángulos con el nombre del objeto subrayado, seguido de dos puntos y el nombre de la clase a la que pertenece (por ejemplo, `miCoche: Coche`). Opcionalmente, pueden incluir una lista de atributos con sus valores actuales.
- **Enlaces:** Líneas que conectan los objetos, representando las relaciones de asociación entre las instancias de las clases.

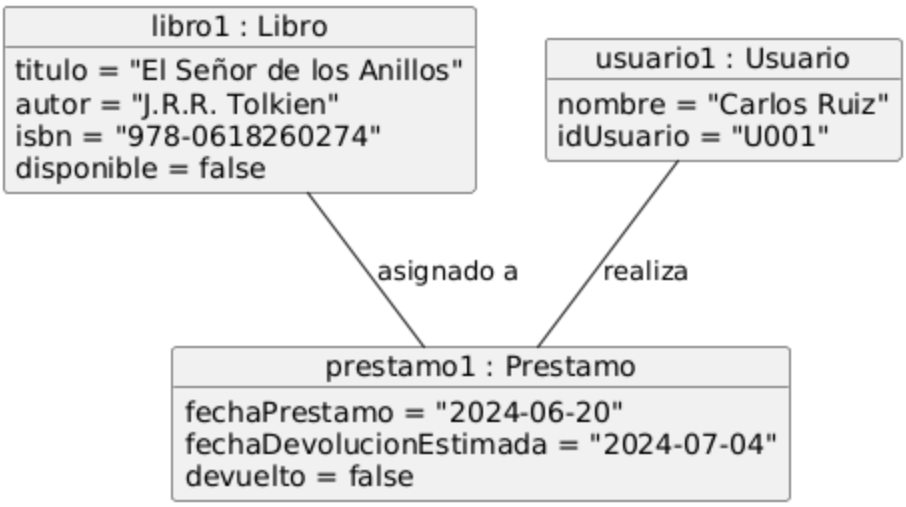


▶ **Haz click aquí para ver el código plantuml**

Observa el diagrama de arriba, que representa una configuración específica de un sistema de pedidos:

1. **Objetos Instanciados:** Vemos `cliente1` como una instancia de `Cliente`, `pedido1` como una instancia de `Pedido`, y `productoA` y `productoB` como instancias de `Producto`. Cada objeto muestra sus atributos con valores específicos en este momento.
2. **Enlaces entre Objetos:**
   - La línea entre `cliente1` y `pedido1` con la etiqueta "realiza" indica que `cliente1` ha realizado el `pedido1`.
   - Las líneas entre `pedido1` y `productoA`, y `pedido1` y `productoB`, con la etiqueta "contiene", muestran que `pedido1` incluye `productoA` y `productoB`.
3. **Cardinalidad (Opcional):** Aunque los diagramas de objetos se centran en instancias, puedes ver la cardinalidad de los enlaces, heredada del diagrama de clases subyacente.

> **Actividad**
> Interpreta el siguiente diagrama de objetos que representa una configuración de un sistema de biblioteca en un momento dado:



▶ **Haz click aquí para ver el código plantuml**

Utiliza un diagrama de objetos cuando:

- Necesitas mostrar un **ejemplo concreto del estado** de un sistema.
- Estás **verificando la estructura** de tu modelo de clases con datos de ejemplo.
- Quieres ilustrar **escenarios de prueba** o configuraciones específicas.
- Deseas comunicar una **instantánea del sistema** en un momento particular.

> **Actividad**
>
> Realiza un diagrama de objetos a partir de la información recabada en el diagrama del primer ejercicio del reto individual.

Ampliación: Diagramas de Componentes

Diagramas de Componentes

Los diagramas de componentes te permiten visualizar la **estructura modular de tu sistema**, mostrando cómo los componentes de software (módulos, bibliotecas, ejecutables, servicios) están organizados y cómo interactúan entre sí. Piensa en ellos como un mapa de las piezas de software construibles y reutilizables de tu aplicación, y cómo encajan para formar el sistema completo. No muestran la implementación interna de un componente, sino sus interfaces y sus relaciones con otros componentes.

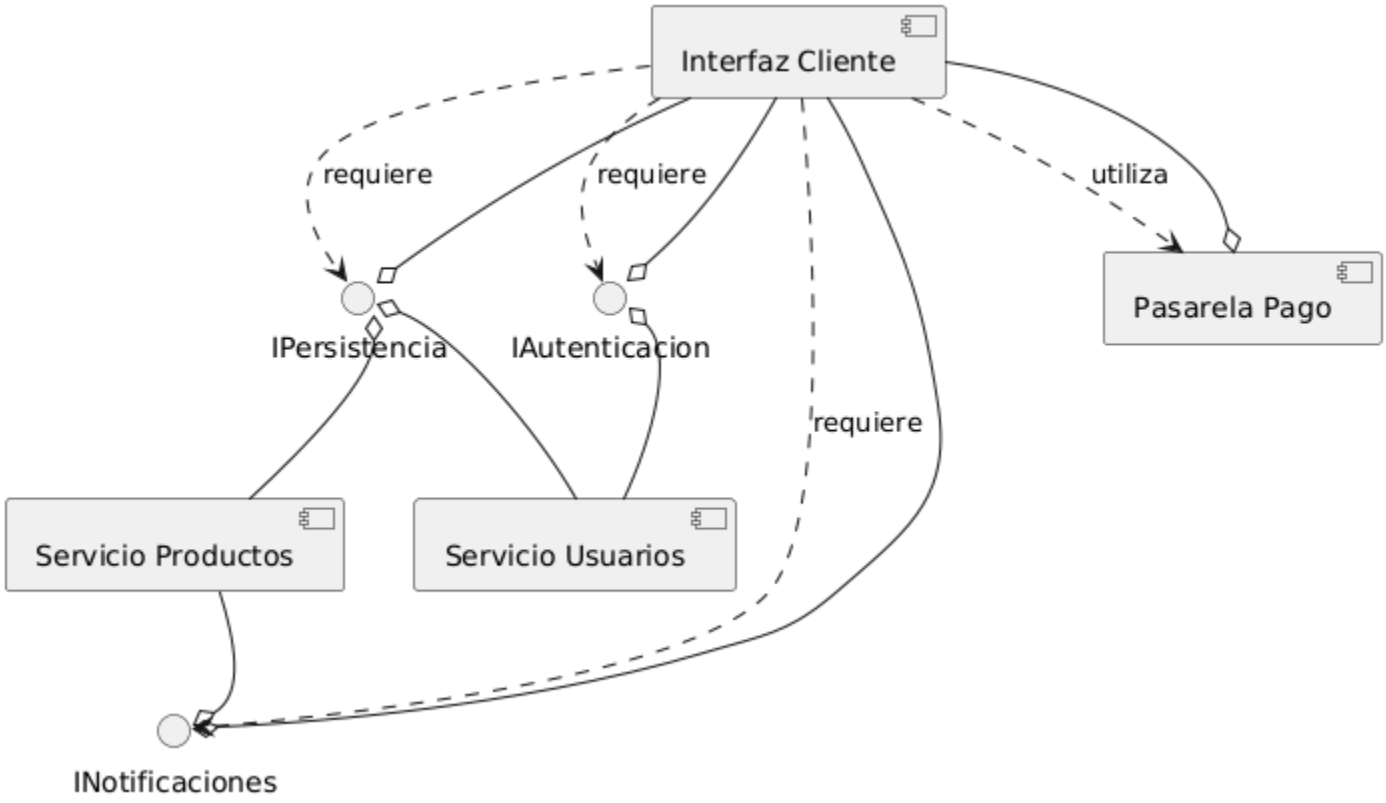
Estos diagramas son especialmente útiles para:

- **Diseño de alto nivel de la arquitectura:** Muestran la división lógica del sistema en unidades funcionales.
- **Gestión de dependencias:** Ayudan a comprender cómo los cambios en un componente pueden afectar a otros.
- **Reusabilidad y modularidad:** Fomentan el diseño de componentes con interfaces bien definidas para facilitar su reutilización.
- **Documentación de la arquitectura:** Proporcionan una vista clara de la estructura del sistema para desarrolladores y stakeholders.

Estructura de un Diagrama de Componentes

Un diagrama de componentes se compone de:

- **Componentes:** Representados como un rectángulo con dos pequeños rectángulos a un lado (tipo "caja") o simplemente un rectángulo con la palabra `<<component>>` en su interior.
- **Interfaces:** Muestran cómo un componente expone sus funcionalidades a otros (interfaces provistas, representadas con una "bola" o círculo) y cómo depende de funcionalidades ofrecidas por otros (interfaces requeridas, representadas con un "socket" o media luna).
- **Dependencias:** Líneas punteadas con una flecha que indican que un componente depende de otro.
- **Conectores (ensamblajes):** La combinación de una interfaz provista con una interfaz requerida, mostrando que un componente utiliza la funcionalidad de otro.

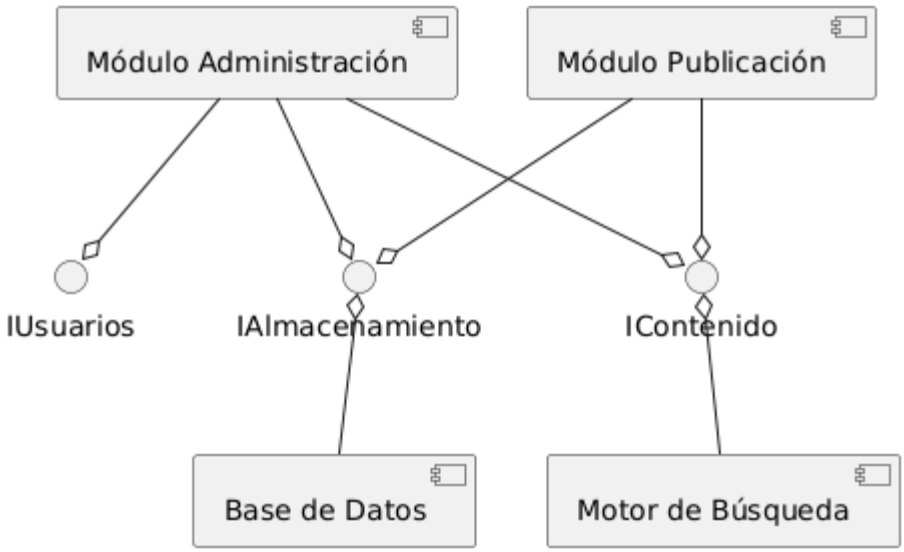


▶ **Haz click aquí para ver el código plantuml**

Observa el diagrama de arriba, que representa la estructura de un sistema de comercio electrónico:

1. **Componentes:** Tenemos `Servicio Usuarios`, `Servicio Productos`, `Pasarela Pago` e `Interfaz Cliente`.
2. **Interfaces:**
   - `IAutenticacion` es una interfaz que el `Servicio Usuarios` **provee** (la "bola"). La `Interfaz Cliente` la **requiere** (el "socket"). La conexión entre la bola y el socket muestra que la `Interfaz Cliente` utiliza la funcionalidad de autenticación del `Servicio Usuarios`.
   - Similarmente, `IPersistencia` es provista por `Servicio Usuarios` y `Servicio Productos`, y requerida por `Interfaz Cliente`.
   - `INotificaciones` es provista por `Servicio Productos` y requerida por `Interfaz Cliente`.
3. **Dependencias (opcionales en PlantUML si hay ensamblaje directo):** Aunque no se muestran explícitamente como líneas punteadas de dependencia en este ejemplo gracias a los ensamblajes, la conexión visual de las interfaces ya implica una dependencia.
4. **Ensamblajes:** Las líneas que conectan la "bola" (interfaz provista) con el "socket" (interfaz requerida) muestran las conexiones directas entre los componentes.

> **Actividad**
>
> Interpreta el siguiente diagrama de componentes que describe un sistema de gestión de contenidos (CMS):



▶ **Haz click aquí para ver el código plantuml**

Utiliza un diagrama de componentes cuando:

- Estás diseñando la **arquitectura modular** de tu sistema.
- Necesitas visualizar las **dependencias entre módulos** y subsistemas.
- Quieres comunicar las **interfaces públicas** de tus componentes.
- Estás planeando la **reutilización de código** o el desarrollo de un sistema basado en servicios.

> **Actividad**
> Realiza un diagrama de componentes que refleje la organización de un instituto, con aulas, despachos, talleres, pizarras, etc.

Ampliación: Diagramas de Despliegue

Diagramas de Despliegue

Los **Diagramas de Despliegue** (Deployment Diagrams) son fundamentales en UML para **visualizar la configuración física del hardware y el software de tu sistema**. Muestran cómo se distribuyen los componentes y artefactos de software (ejecutables, bibliotecas, bases de datos) en los nodos físicos (servidores, dispositivos, etc.) en un entorno de producción o desarrollo.

Imagina un diagrama de despliegue como un plano de tu infraestructura de TI, donde cada elemento físico es una "caja" y dentro de esas cajas se encuentran los programas, archivos y datos que conforman tu aplicación.

Estos diagramas son muy útiles para:

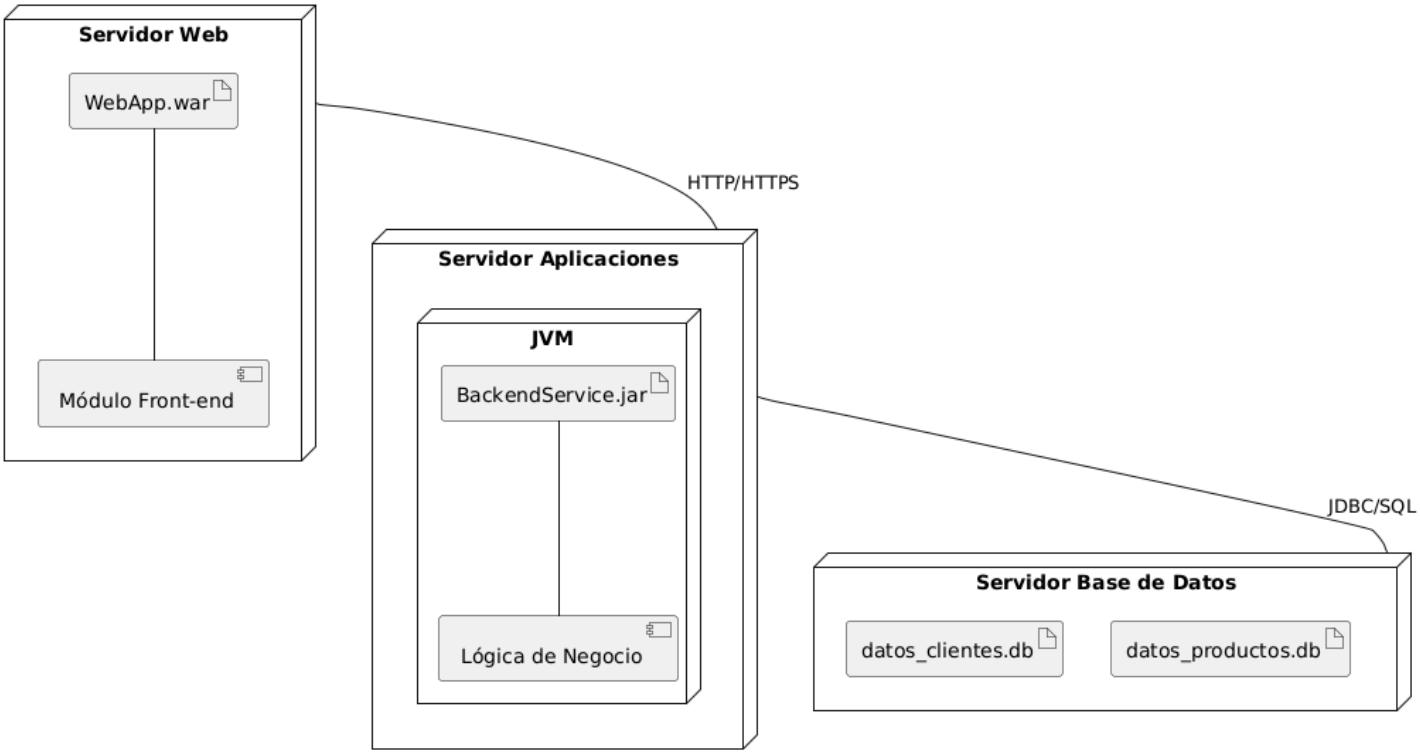
- **Planificación de la infraestructura:** Ayudan a definir la arquitectura física del sistema.
- **Identificación de cuellos de botella:** Permiten visualizar dónde se pueden concentrar las cargas de trabajo.
- **Gestión de la configuración:** Muestran qué software reside en qué hardware.
- **Documentación del entorno de producción:** Proporcionan una vista clara de cómo se despliega el sistema.

Estructura de un Diagrama de Despliegue

Un diagrama de despliegue se compone de:

- **Nodos:** Representados como cubos 3D, representan recursos computacionales físicos (servidores, estaciones de trabajo, dispositivos móviles, etc.) o lógicos (entornos de ejecución como JVMs o contenedores). Pueden contener otros nodos.
- **Artefactos:** Representados como un documento con una esquina doblada, son las unidades físicas del software (archivos ejecutables, librerías, archivos de configuración, scripts, bases de datos). Residen en los nodos.
- **Asociaciones:** Líneas que conectan los nodos, mostrando las relaciones de comunicación entre ellos (por ejemplo, una conexión de red). Pueden llevar estereotipos para indicar el tipo de conexión (TCP/IP, HTTP, etc.).

- **Componentes (opcional pero común):** A menudo se incluyen componentes dentro de los artefactos o nodos para mostrar qué componentes lógicos están siendo desplegados.

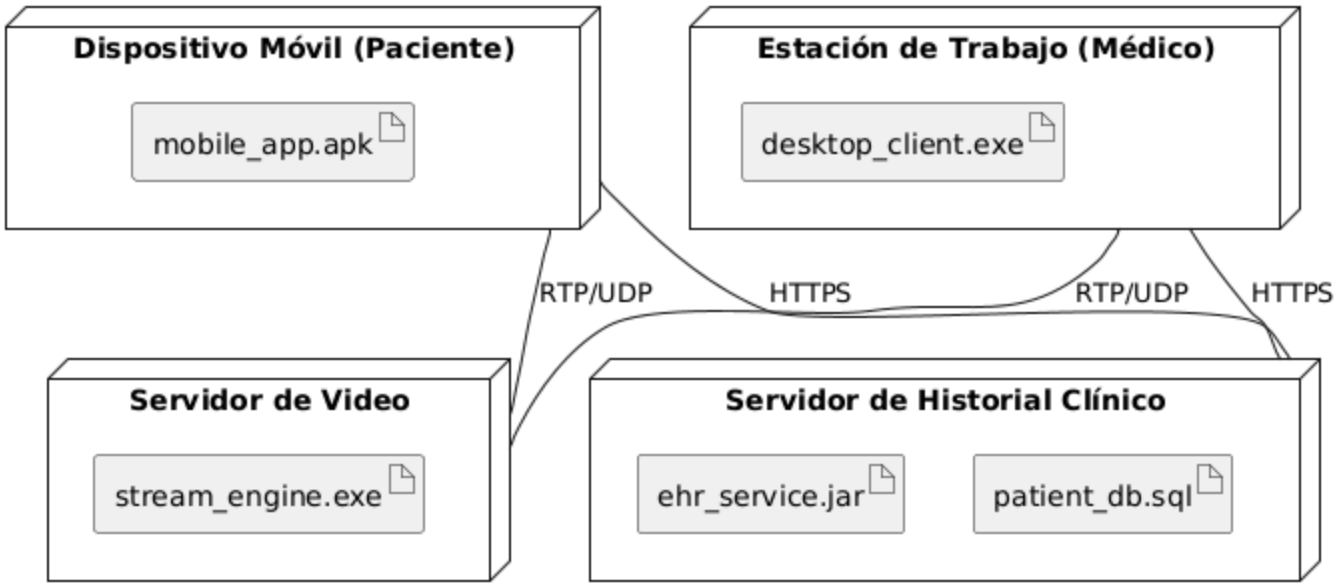


▶ **Haz click aquí para ver el código plantuml**

Observa el diagrama de arriba, que representa el despliegue de una aplicación web:

1. **Nodos Físicos:** Tenemos `Servidor Web`, `Servidor Aplicaciones` y `Servidor Base de Datos`, representados como cubos.
2. **Nodo Anidado:** Dentro del `Servidor Aplicaciones`, hay un nodo anidado `JVM`, que representa un entorno de ejecución.
3. **Artefactos:**
   - En `Servidor Web`, está `WebApp.war` (el archivo de la aplicación web).
   - En `JVM` (dentro de `Servidor Aplicaciones`), está `BackendService.jar` (el archivo del servicio backend).
   - En `Servidor Base de Datos`, tenemos `datos_clientes.db` y `datos_productos.db` (archivos de base de datos).
4. **Componentes (opcionales):** Se han incluido componentes como `Módulo Front-end` y `Lógica de Negocio` para mostrar qué lógica de software está contenida dentro de los artefactos.
5. **Asociaciones (Conexiones):**
   - `WebServer` se conecta a `AppServer` usando "HTTP/HTTPS".
   - `AppServer` se conecta a `DBServer` usando "JDBC/SQL".

> **Actividad**
> Interpreta el siguiente diagrama de despliegue que muestra la infraestructura de un sistema de telemedicina:



▶ **Haz click aquí para ver el código plantuml**

Utiliza un diagrama de despliegue cuando:

- Necesitas mostrar la **arquitectura física** de tu sistema.
- Estás planificando la **distribución de software** en diferentes servidores o dispositivos.
- Deseas documentar el **entorno de producción** o de prueba.
- Quieres visualizar las **conexiones y protocolos de red** entre los nodos.

> **Actividad**
> Crea un diagrama de despliegue a partir de la información recabada en el ejercicio 1 del reto individual.