Tema 4 - Introducción a UML

1. Introducción

Ver PDF adjunto.

1ª PARTE. DIAGRAMAS DE CLASES

2. Orientación a objetos y modelado de los mismos

2.1 El objeto y las clases

Un objeto es una entidad identificable del mundo real. Puede tener una existencia física (un coche, un libro) o no tenerla (un texto de ley). *Identificable* significa que el objeto se puede designar. Por ejemplo:

- Mi coche Ford Fiesta rojo
- Mi libro sobre UML 😃
- El artículo 7 de la LOMLOE

En UML todo objeto tiene una estructura (un conjunto de atributos) y un comportamiento (un conjunto de métodos). Un atributo es una variable destinada a recibir un valor. Un método es un conjunto de instrucciones que toman unos valores de entrada y modifican los valores de los atributos.

Incluso los objetos del mundo real son percibidos siempre como dinámicos. Así, en UML, un coche es un objeto que puede acelerar. **Todo sistema concebido en UML está compuesto por objetos que interactúan entre sí, realizando operaciones propias de su comportamiento**.

> Un grupo de alumnos es un sistema de objetos (con perdón) que interactúan entre sí, y donde cada objeto posee su propio comportamiento.

2.1.1 Abstracción

La abstracción es una simplificación indispensable para el proceso de modelado. Un objeto UML es una abstracción de un objeto del mundo real de acuerdo con las necesidades del sistema de la cual sólo se tienen en cuenta los elementos esenciales.

2.1.2 Clases de objetos

Un conjunto de objetos similares, es decir, con la misma estructura y comportamiento, y constituidos por los mismos atributos y métodos, forman una clase de objetos. La estructura y el comportamiento pueden entonces definirse en común en el ámbito de clase.

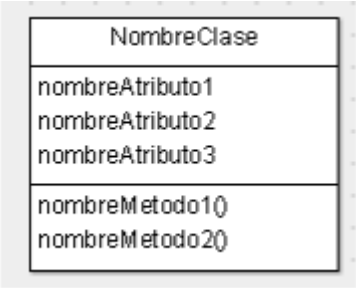
Todos los objetos de una clase, llamada también instancia de clase, se distinguen por tener una identidad propia y sus atributos le confieren valores específicos.

> El conjunto de personas constituye la clase Persona, que posee la estructura y comportamiento de la siguiente imagen.

> Recordemos que el nombre de las clases se escribe en singular y está siempre formado por un nombre común precedido o seguido de uno o varios adjetivos que lo califican. Dicho nombre es representativo del conjunto de objetos que forman la clase y representa la naturaleza de las instancias de una clase.

> En UML, para nombrar las clases se utiliza la notación UpperCamelCase

En UML las clases son representadas mediante una caja dividida en tres partes: nombre de la clase, atributos y operaciones o métodos.



Los atributos contienen información de los objetos. **El conjunto de atributos forma la estructura del objeto.**

Los métodos se corresponden con los servicios ofrecidos por el objeto y pueden modificar el valor de los atributos. **El conjunto de métodos forma el comportamiento del objeto**.

> El número de atributos y métodos varía de acuerdo con la clase. No obstante, se desaconseja emplear un número elevado de ellos ya que, en general, éste refleja una mala concepción de la clase.

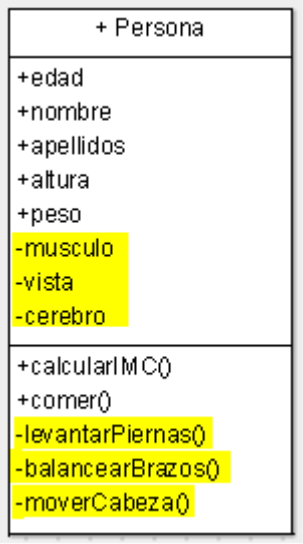
Esta es la forma más simple de representación de clase, porque no hace aparecer las características de los atributos y de los métodos (tipo de dato, argumentos, tipo de retorno, ...), a excepción de su nombre. Se utiliza a menudo en las primeras fases del modelado (análisis).

2.1.3 Encapsulación

La encapsulación consiste en ocultar los atributos y métodos del objeto a otros objetos. En efecto, algunos atributos y métodos tienen como único objetivo tratamientos internos del objeto y no deben estar expuestos a los objetos exteriores. Una vez encapsulados, pasan a denominarse atributos y métodos privados del objeto

La encapsulación es una abstracción, ya que se simplifica la representación del objeto con relación a los objetos externos. Esta representación simplificada está formada por atributos y métodos públicos del objeto

> *Al andar, las personas efectúan diferentes movimientos como pueden ser levantar las piernas, balancear los brazos o mover la cabeza. Esos movimientos son internos al funcionamiento de la persona y no tiene por qué ser conocidos en el exterior. Son métodos privados. Las operaciones acceden a una parte interna de la persona: sus músculos, su cerebro y su vista. La parte interna se representa en forma de atributos privados.*



UML, al igual que la mayoría de lenguajes modernos orientados a objetos, introduce tres posibilidades de encapsulación

- El atributo o el método privado: la propiedad no se expone fuera de la clase, ni tampoco dentro de sus *subclases* (hablaremos de ellas en el apartado de herencia).
- El atributo o el método protegido: la propiedad sólo se expone en las instancias de la clase y sus subclases.
- La encapsulación de empaquetado: la propiedad sólo se exponen en las instancias de clases del mismo empaquetado (en Java, quiere decir que son accesible desde todas las clases del mismo paquete).

La noción de propiedad privada conduce a establecer una diferencia entre las instancias de una clase y las de sus subclases. Esta diferencia está vinculada a aspectos bastante sutiles de la programación con objetos. La encapsulación de empaquetado, por su parte, procede del lenguaje Java.

Cada uno de los tipos de encapsulado tiene un signo que se coloca delante del nombre de la

Encapsulado	Signo	Descripción
Público	+	Visible para todos
Protegido	#	Visible en las subclases de la clase
Privado	-	Visible sólo en la clase
Empaquetado	~	Visible sólo en las clases del mismo paquete

2.1.4 La noción de tipo

De manera general, llamamos variable a cualquier elemento que pueda tomar un valor, es decir, a cualquier atributo, parámetro o incluso a los valores de retorno de un método.

El tipo es una especificación aplicada a una variable. Consiste en fijar el conjunto de valores posibles que la variable puede tomar. Dicho conjunto puede ser:

- Un tipo estándar o primitivo, como el conjunto de enteros, cadenas de caracteres, booleanos o reales.

- Una clase, en cuyo caso la variable debe contener una **referencia a una instancia de la misma**.

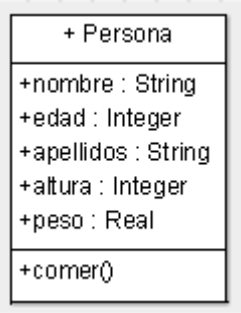
En UML, los tipos primitivos o estándar son:

- `Integer`, para números enteros: `1, 3, 10`.
- `String`, para las cadenas de caracteres: `Coche`, `Hola`
- `Boolean`, para los valores booleanos: `true`, `false`.
- `Real`, para los números reales: `3.1415`.

A la hora de escoger un tipo de clase para un atributo de una clase, tendremos las siguientes situaciones:

- Si el tipo de clase forma parte de nuestro sistema, diremos que tiene una **asociación**. Hablaremos de ellas más adelante.
- Si no forma parte de nuestro sistema (una biblioteca externa), lo ideal será siempre que sea posible un tipo interfaz (las veremos más adelante); y si no es posible, escogeremos la clase en cuestión.

La forma de representar el tipo de un atributo es la siguiente:



2.1.5 Firma de un método

Un método de una clase puede tomar parámetros y devolver (o no) un resultado. Los parámetros son valores transmitidos:

- En la ida, al enviar un mensaje que llama a un método.
- En el retorno de la llamada del método.

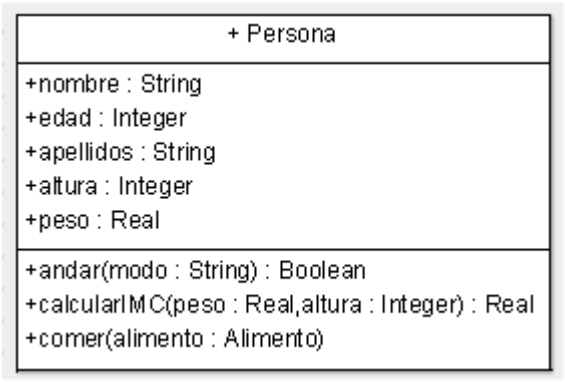
El resultado es un valor transmitido al objeto que efectúa la llamada de dicho método cuando ésta la devuelve.

Como hemos visto anteriormente, tanto los parámetros como el resultado pueden tener tipos. El conjunto constituido por el nombre del método, los parámetros con su nombre y su tipo, así como el tipo de resultado, se conoce como **la firma del método**. Una firma adopta la siguiente forma:

```
<nombreMetodo> (<dirección> <nombreParametro> : <tipo>, ...) : <tipoResultado>
```

En UML, es posible indicar la dirección en la cual se transmite el parámetro, colocando delante *in* (entrada), *out* (salida) o *inout* (entrada y salida).

Si no se especifica nada, se entiende que por defecto es *in*.



2.1.6 Atributos y métodos de clase (que no de objeto o instancia)

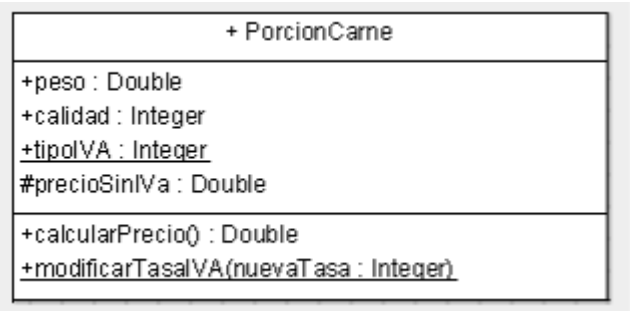
Las instancias de una clase contienen un valor específico para cada uno de sus atributos. Este valor, por tanto, no se comparte con el conjunto de instancias. En algunos casos, es preciso utilizar atributos cuyo valor es común a todos los objetos (instancias) de una clase. Estos atributos son conocidos como *atributos de clase*, porque están vinculados a la clase y lo a las instancias.

> En muchas herramientas UML, no se utilizan los términos "atributo y método de clase". Estas herramientas dan preferencia a la denominación "atributos o métodos estáticos", denominación propia de lenguajes como Java o C++. En StarUML, para indicar que un atributo es abstracto hay que marcar la casilla *isStatic*.

Los atributos y métodos estáticos se representan subrayando su nombre. Pueden estar encapsulados y poseer un tipo. Además, se recomienda asignarles un valor determinado.

> Por ejemplo, en un negocio donde se vendan productos tecnológicos, podríamos almacenar el IVA como un atributo estático, ya que todos comparten el valor.

Dentro de una clase, también puede existir uno o varios métodos de clase vinculados a la misma. Para llamar a un método de clase, hay que *enviar un mensaje a la propia clase* y no a una de sus instancias. Estos métodos sólo manipulan los atributos de clase.



> *El paso de mensajes entre objetos es la forma que tienen de interactuar. De forma abstracta, se llama mensaje a una porción de información que un proceso (u objeto) emisor envía a un destinatario (el cual puede ser otro proceso, un actor o un objeto). El modelo de paso de mensajes es el que define los métodos y funciones para poder llevar a cabo el envío de un mensaje de un proceso emisor a un destinatario. Supone un enfoque opuesto al paradigma tradicional en el cual los procesos, funciones y subrutinas sólo podían ser llamados directamente a través de su nombre.* Para más info, pulsa aquí.

> Los atributos y métodos estáticos **NO SE HEREDAN**. La herencia se aplica a la descripción de las instancias, calculada a través de la unión de la estructura y del comportamiento de la clase y de sus superclases. Una subclase puede acceder a un atributo o a un método estático de una de sus superclases, pero no los hereda. De haber herencia, tendríamos tantos ejemplares de atributos o métodos como subclases poseyera la clase que los introdujo. Recordamos también que una subclase puede acceder a un atributo o a un método de clase de una de sus superclases, a condición de que no se haya declarado como privado.

2.2 Especialización y generalización: herencia

Las clases pueden definirse también como subconjuntos de otras clases, con las que comparten propiedades.

Hablamos entonces de **subclases** de otras clases que, por tanto, constituyen **especializaciones** de esas otras clases.

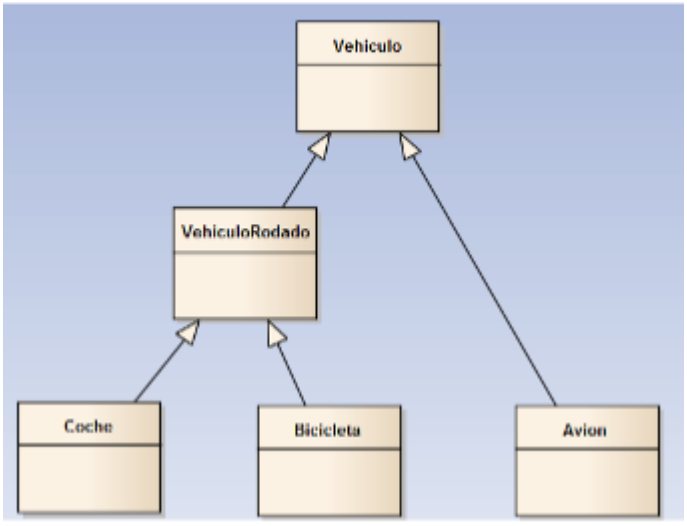
> *La clase coche es una subclase de la clase de los vehículos*

La **generalización** es la inversa de la especialización. Se una clase es una especialización de otra, ésta última es una generalización de la primera. Es su **superclase**.

> *La clase de los vehículos es una superclase de la clase de los coches*.

La relación de especialización puede tener más de un nivel, dando lugar a una jerarquía de clases.

> *La clase de los vehículos se concreta en vehículos rodados y aviones; y los vehículos rodados, a su vez, en coches y bicicletas*



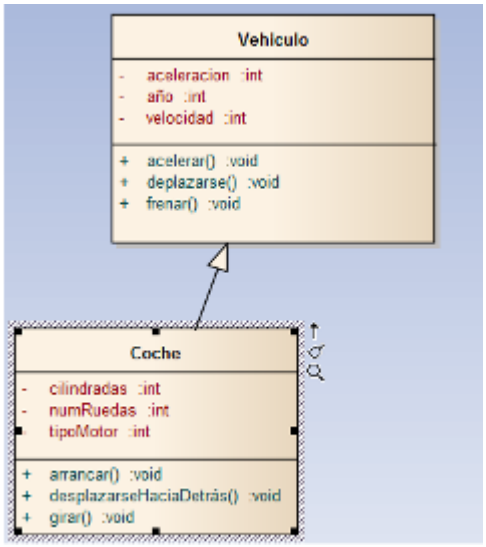
> **La forma de especificar este tipo de relaciones en UML es mediante una flecha de punta cerrada (la punta tendrá el color de fondo de las clases); la flecha irá desde la subclase apuntando hacia la superclase**.

2.3 Herencia

La herencia es la propiedad que hace que una subclase se beneficie de la estructura y el comportamiento de su superclase. Es por tanto, una consecuencia de la especialización.

Las instancias de una subclase son, así mismo, instancias de la superclase, y por consiguiente, además de la estructura y el comportamiento introducidos en la subclase, se benefician también de la estructura y el comportamiento definido por la subclase.

En el siguiente ejemplo, la clase `Coche` es una especialización de la clase `Vehículo`, y su estructura y comportamiento se describe como una combinación de ambas.



> Aunque la herencia es una consecuencia de la especialización, se suele usar más el término *hereda* que *especializa*.

2.4 Clases concretas y clases abstractas

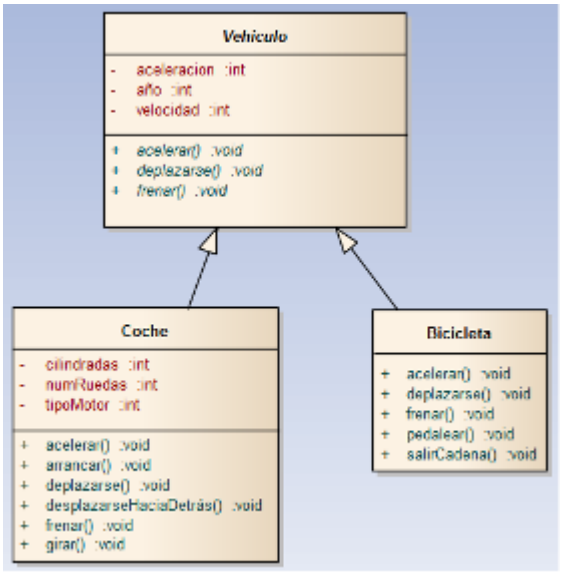
Se revisamos la jerarquía de herencia de los coches y bicicletas, vemos que hay dos tipos de clases.

Por un lado, aquellas que poseen instancias, como `Coche` y `Bicicleta`, llamadas **clases concretas**.

Por otro lado, las que no poseen directamente instancias, como la clase `Vehículo`. En efecto, aunque en el mundo hay miles de coches, aviones, ... el concepto de vehículo es abstacto. No basta para definir completamente un vehículo. Por ello decimos que la clase *`Vehículo`* es **abstracta**.

La finalidad de las clases abstractas es poseer subclases concretas, pero dar un marco común a todas ellas. De esta manera, la clase abstracta fija los atributos y operaciones mínimas que debe tener cualquier clase que la concrete.

> *En UML cualquier elemento (clase, atributo u operación) que sea abstracto se representa con letra cursiva*.



2.5 Polimorfismo

El polimorfismo significa que una clase (generalmente abstracta, aunque no tiene porque serlo) representa un conjunto formado por objetos diferentes, ya que éstos son instancias de subclases diferentes. Cuando se llama a un método del mismo nombre, esta diferencia se traduce en comportamientos distintos (excepto en los casos en los que el método es común y las subclases lo han heredado de la superclase).

> Si nos fijamos en el diagrama de la imagen anterior, el método `acelerar()` tiene un comportamiento diferente según si el vehículo es una instancia de `Coche` o de `Bicicleta`. Si consideramos la clase `Vehículo` en su totalidad, tenemos un conjunto de vehículos que reaccionan de distinta manera al activarse el método `acelerar()`.

En Java, lo podríamos traducir así:

```
public abstract class Vehiculo {

    // Resto del código

    public abstract void acelerar();

}


public class Coche extends Vehiculo {

    // Resto del código

    @override
    public void acelerar() {
        System.out.println("Incrementando velocidad");
    }

}
```

```java
public class Bicicleta extends Vehiculo {

    // Resto del código

    @override
    public void acelerar() {
        System.out.println("Dale más fuerte a los pedales, campeón");
    }

}


public class Main {

    public static void main(String[] args) {

        Vehiculo v1 = new Coche();
        Vehiculo v2 = new Bicicleta();

        v1.acelerar();  // Salida: Incrementando velocidad
        v2.acelerar();  // Salida: Dale más fuerte a los pedales, campeón

    }


}
```

2.6 Interfaces

A veces sucede que cuando hacemos diagramas sobre sistemas extensos, nos damos cuenta que algunas clases comparten cierto comportamiento. Esto nos deja algunas alternativas.

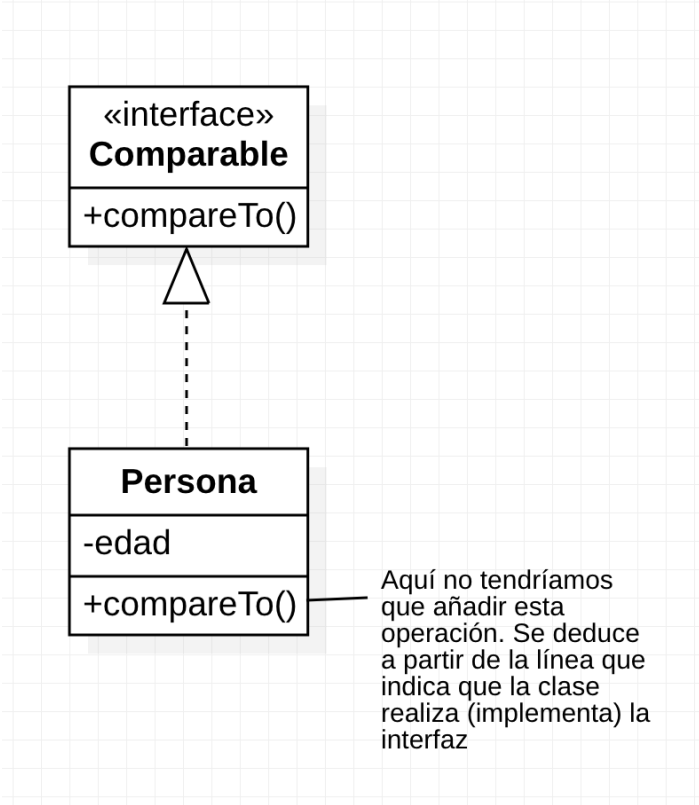
- Una de ellas sería la herencia, pero es posible que esas clases ya hereden de otra clase (y muchos de los lenguajes orientados a objetos no permiten herencia múltiple).
- Otra de ellas se nos da cuando se comparte únicamente comportamiento, pero no tienen ningún atributo en común.

Sea como fuere, podemos utilizar en ambos casos una **interfaz**, que nos permite especificar algunas de las operaciones de una clase.

Una **interfaz** es un contrato que una clase se compromete a cumplir. El compromiso es una serie de operaciones que debe implementar.

En UML tenemos varias formas de representar una interfaz, pero una de las más usuales es a través de un rectángulo similar a las clases, con las siguientes características:

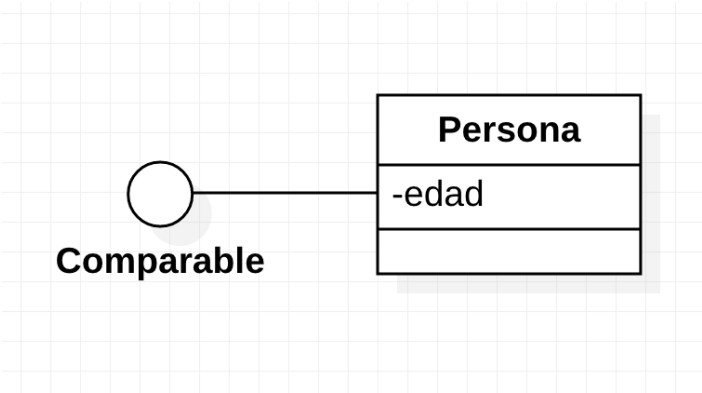
- El apartado de las propiedades estaría vacío
- Solamente tendría contenido real en el de las operaciones.
- Se nombran *igual* que las clases, pero se le añade el estereotipo `<<interface>>`.



La asociación entre una interfaz y la clase que la implementa se llama *realización* (*realizes*) y se representa mediante una flecha con la línea punteada y la punta cerrada.

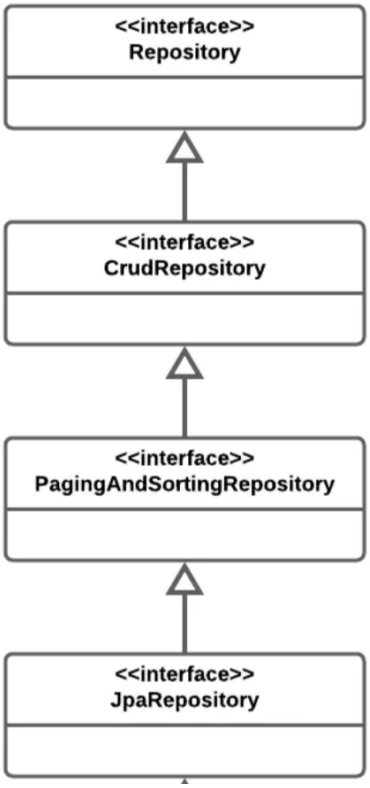
> Si pensamos en código Java, las interfaces permitirán crear referencias de variables donde el tipo de dato sea la propia interfaz. Y así poder invocar a métodos que están definidos en la interfaz. Según el diagrama anterior podríamos tener el siguiente código:
>
> ```
> Comparable per = new Persona();
> ```
>
> o bien usarlo en una como argumento en una llamada a un método.
>
> ```
> void metodo(Comparable c) { ... }
> metodo(per);
> ```

También tenemos la forma abreviada de representar una interfaz, a partir de un círculo. StarUML nos permite ambas representaciones (de hecho, inicialmente usa la forma abreviada). En el caso abreviado, la línea será continua y sin punta de flecha.



Una clase puede realizar tantas interfaces como quiera. De esta manera, algunos lenguajes como Java consiguen un comportamiento parecido a la herencia múltiple, sin tener que dirimir con los problemas que ésta origina.

Por otro lado, también podemos tener relaciones de especialización entre dos o más interfaces, de forma que se llegue a tener una jerarquía de herencia entre ellas.



2.7 Enumeraciones

Hemos visto cómo en UML podemos representar propiedades de *tipo primitivo* (`String`, `Integer`, ...). También veremos que, gracias a las asociaciones, en una clase podremos tener instancias de otras clases, es decir, variables cuyo tipo de dato sea otra clase (en una instancia de `Pedido` es posible que tengamos la referencia a una instancia de `Cliente`).

En algunas ocasiones puede resultar útil para nosotros la creación de nuestros propios tipos de dato. Sobre todo, para aquellos casos donde lo que queremos es crear tipos cuyos posibles valores sean un conjunto cerrado.

> Un ejemplo clásico serían los posibles estados de un pedido: `INICIADO`, `CONFIRMADO`, `PREPARADO`, `ENVIADO` y `ENTREGADO`.

Posiblemente esta situación la podamos manejar con un atributo de tipo `String` (o del tipo cuyos valores queremos representar), pero UML (y posteriormente, los lenguajes de programación) nos proporcionan el concepto de **enumeración**. Una enumeración no es más que un tipo de dato definido por el usuario que consta de una serie de constantes de cadena (en Java, internamente, representadas también por un ordinal numérico) delimitadas por comas.

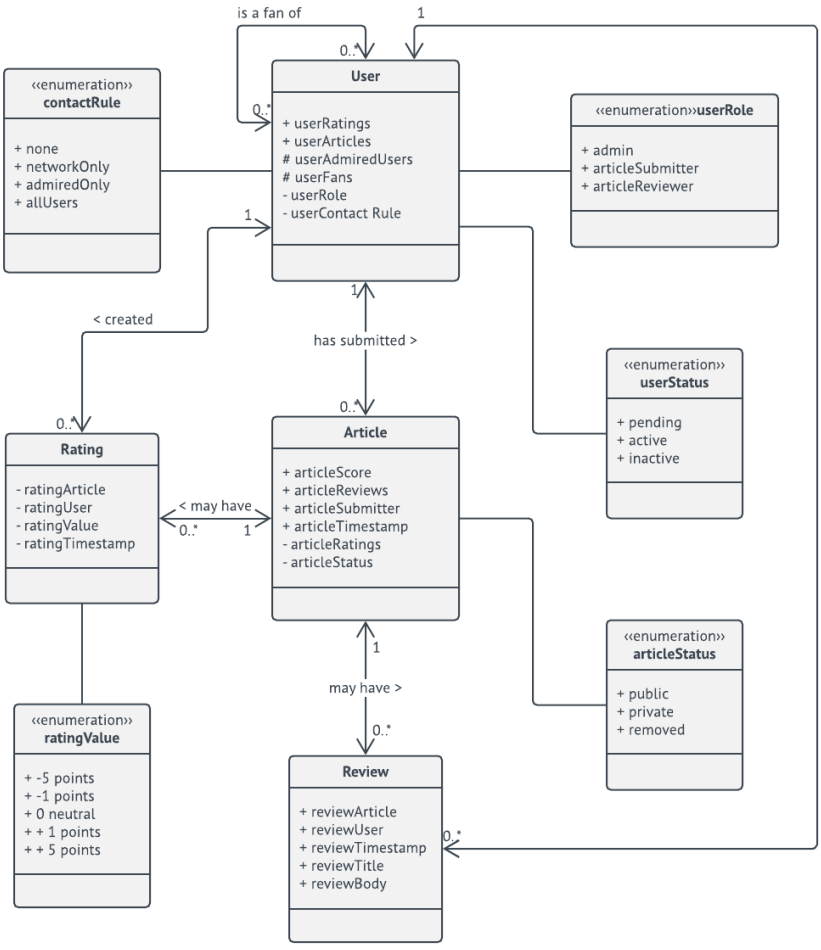
```
public enum OrderStatus {

    INITIALIZED, COMMITED, PREPARED, SHIPPED, DELIVERED

}
```

> En Java, las enumeraciones pueden ser casi tan potentes como las clases (de hecho son clases que heredan de `Enum<T>`), incluyendo también métodos, atributos como constructores.

En UML podemos representar una enumeración de una forma muy parecida a una clase. Se utiliza un rectángulo igual, con el nombre de la enumeración arriba. Habitualmente, dentro del apartado de las propiedades aparece el conjunto de posibles valores que puede tomar la enumeración. También se podrían incluir las propiedades y las operaciones, si las tuviera. **Para diferenciar las enumeraciones de las clases, se le añade el estereotipo <<enumeration>> encima del nombre de la clase**



3. Asociaciones entre objetos

> Lo que se va a explicar en este apartado es de aplicación para diagramas de clases que se realizan en la **fase de análisis** del desarrollo de SW. Un diagrama de clases de la *fase de diseño* en adelante debería incluir algunos elementos más, además de lo reflejado aquí.

3.1 Vínculos entre objetos

En el mundo real, los objetos suelen estar vinculados entre sí. Dicho vínculo surge de una asociación entre ellos:

- Una persona y su padre o su madre.
- Un trabajador y la empresa en la que trabaja.
- Un producto y la categoría a la que pertenece.

En UML, estos vínculos se expresan mediante asociaciones, de igual modo que los objetos se describen mediante clases. Una asociación vincula clases, y por tanto, sus instancias.

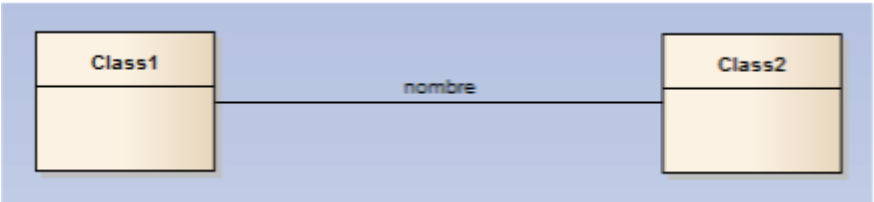
Las asociaciones **tienen un nombre**, y como ocurre con las clases, este es un reflejo de los elementos de la asociación.

- La asociación *progenitor* entre `Padre` y `Descendiente`
- La asociación *trabaja* entre `Trabajador` y `Empresa`.
- La asociación *pertenece* entre `Categoría` y `Producto`.

> Las asociaciones que hemos estudiado hasta el momento en forma de ejemplos establecen un vínculo entre dos clases, por lo que reciben el nombre de **asociaciones binarias**. Las asociaciones que vinculan tres clases se denominan asociaciones **ternarias** y, en general, aquellas que vinculan n clases reciben el nombre de asociaciones **n-arias**. *En la práctica, la gran mayoría de asociaciones son binarias y las cuaternarias y superiores apenas se utilizan*.

3.2 Representación de las asociaciones entre clases.

La representación gráfica de una asociación binaria consiste en una linea continua que une las dos clases cuyas instancias se vinculan. Las clases se sitúan en los extremos de la asociación. El nombre de la asociación se indica encima de la línea

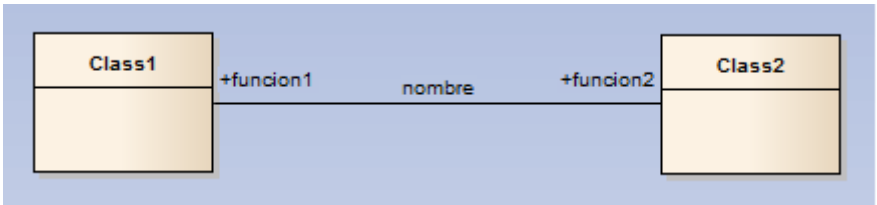


> Para señalar el sentido de lectura del nombre de la asociación con respecto al nombre de las clases, la linea puede contener puntas de flecha

Los extremos de una asociación pueden recibir un nombre. Dicho nombre es representativo de la función (o rol) que desempeñan en la asociación las instancias de la clase correspondiente.

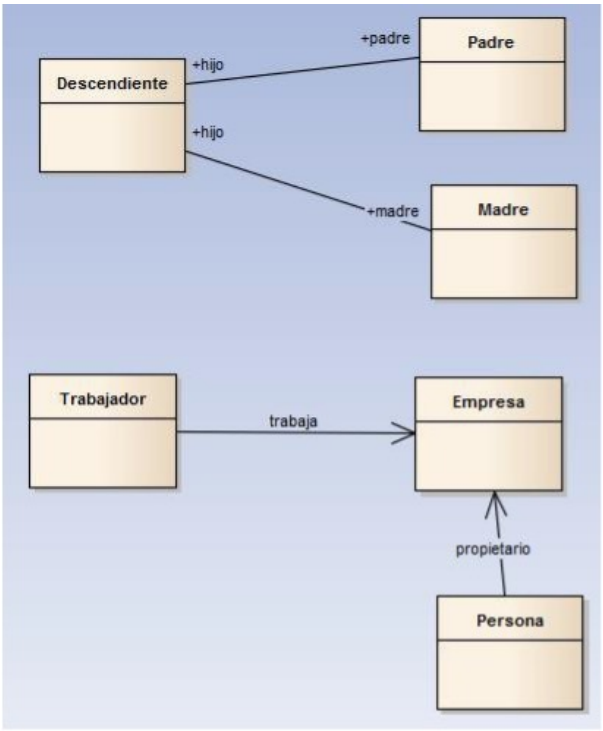
> Así, a la hora de traducir un diagrama a código, una clase tendrá un atributo cuyo tipo será la clase situada en el otro extremo de la asociación.

Los extremos de la asociación pueden tener un modificador de acceso o visibilidad, de forma que puede ser pública o estar encapsulada de alguna de las maneras ya vistas. Cuando se especifican las funciones, muchas veces no es preciso indicar el nombre de la asociación, ya que éste suele ser el mismo que el de una de las funciones.

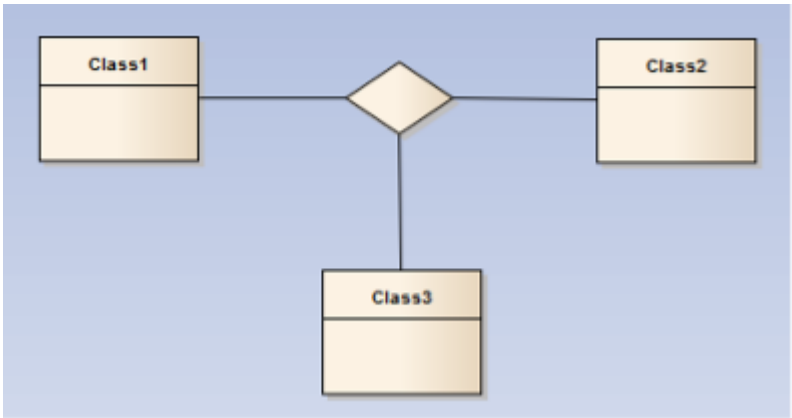


Las asociaciones también pueden incluir una dirección de navegación. Si no se especifica nada, la navegación es bidireccional (desde cualquier extremo al opuesto). En la práctica, puede que no nos convenga tener este tipo de navegación, queriendo que sea ésta unidireccional. Para representar la unidireccionalidad, añadiremos una punta de flecha a uno de los extremos de la asociación, permitiendo así leer desde el origen de la asociación (sin punta de flecha) hasta el destino de la misma (el que incluye la punta de flecha).

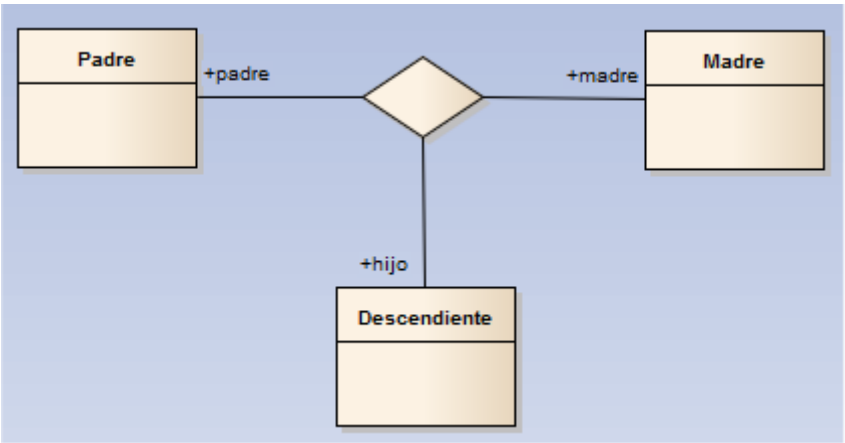
Algunos ejemplos de asociaciones con roles o dirección:



Cuando la asociación no es binaria sino ternaria, es decir, que la asociación afecta a la vez a tres clases, se añade un rombo que une las diferentes clases:



La siguiente sería un ejemplo de asociación ternaria, en la cual le podemos añadir un rol a cada extremo de la asociación:



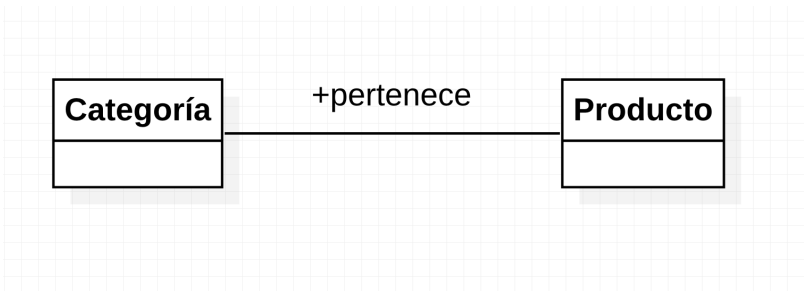
También podrían existir asociaciones cuaternarias (cuatro clases), quinarias (cinco clases), o *n-arias*.

> En general, las asociaciones que unen más de dos clases son **muy difíciles** de manejar, y en la medida de lo posible, deberíamos buscar alternativas que nos permitan adaptarlas a asociaciones binarias.

3.3 La cardinalidad o multiplicidad de las asociaciones

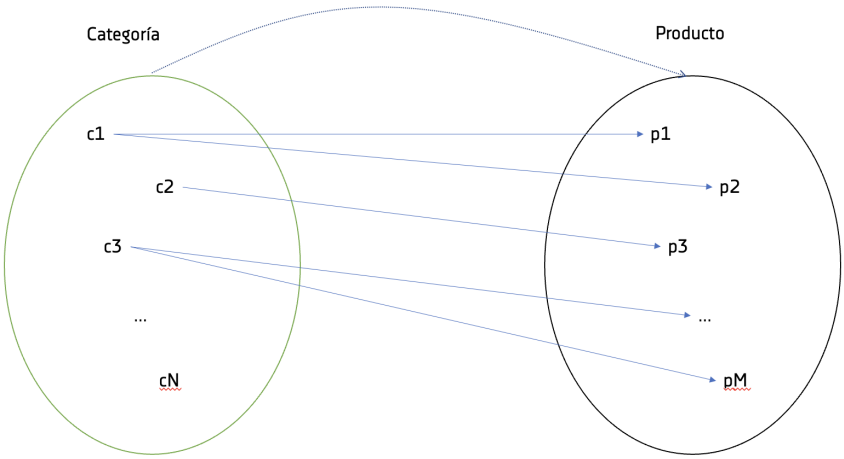
La cardinalidad situada en un extremo de una asociación indica a cuántas instancias de la clase situada en ese mismo extremo está vinculada una instancia de la clase situada en el extremo opuesto.

Sea el siguiente ejemplo de asociación entre las clases `Categoría` y `Producto`:



Según el aserto anterior, la cardinalidad en el extremo de la asociación que hay junto a `Categoría` indica cuántas instancias de `Categoría` pueden estar vinculadas a una instancia de `Producto`.

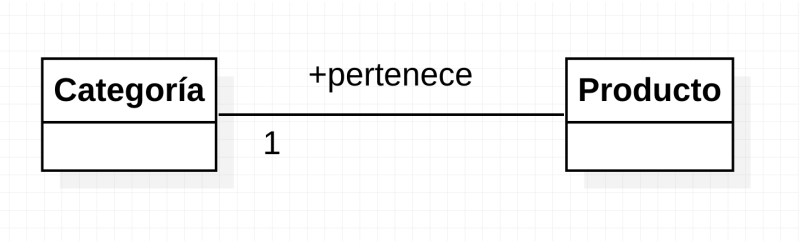
Veamos el siguiente gráfico:



Según el mismo:

- Cada instancia de `Producto` recibe una única *flecha* desde `Categoría`.
- Todas las instancias de `Producto` reciben, al menos, una *flecha* de `Categoría`.

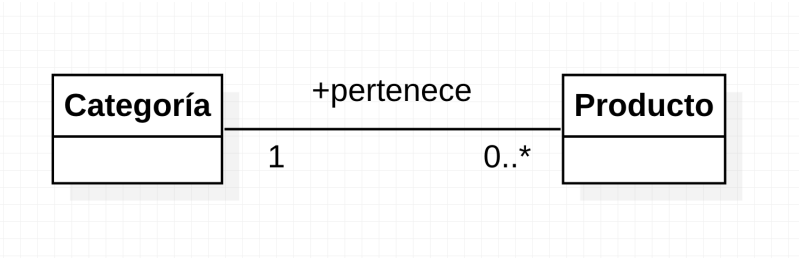
Esto implica que tendríamos que añadir, en ese extremo de la asociación, la multiplicidad `1`, tal y como se representa en el siguiente diagrama:



Si revisamos ahora el extremo opuesto, el de `Producto`, podemos comprobar que:

- De cada instancia de `Categoría` puede salir, o no, una flecha hacia producto.
- De una instancia de `Categoría` puede salir más de una flecha hacia instancias diferentes de `Producto`.

Esto implica que la multiplicidad de este extremo sería `0..*`, y se representaría así:



> La multiplicidad de esta asociación la podríamos *leer así*: una categoría puede estar asociada con cero o más productos, y un producto puede estar asociado a una y sólo una categoría.

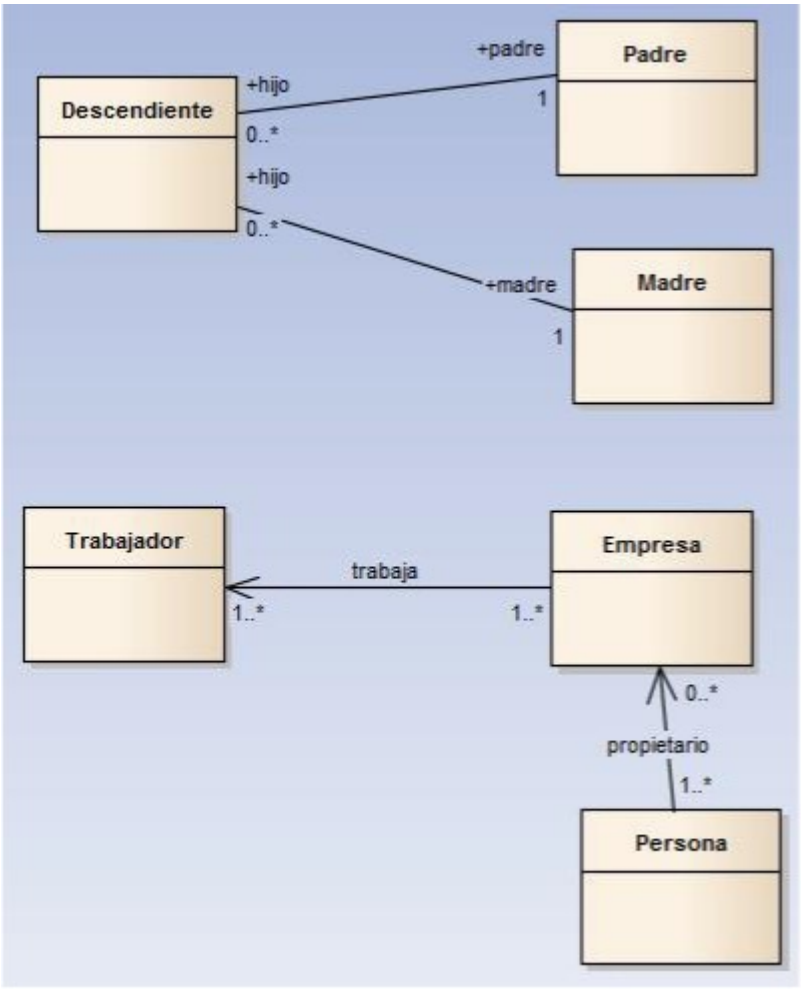
Cuando exista una multiplicidad mínima y una máxima, se expresarán ambas, separadas por dos puntos seguidos.

> Habitualmente, cuando no se indica la cardinalidad o multiplicidad de un extremo, se entiende que es 1. Pero para evitar ambigüedad, añadiremos siempre la multiplicidad de cada extremo de la asociación.

La siguiente tabla describe la sintaxis de especificación de multiplicidades:

Especificación	Multiplicidad
0..1	Cero o una vez
1	Únicamente una vez
0..*	De cero a varias veces
1..*	De una a varias veces
N	N veces, siendo N un número concreto (2, 3, 5 ...)
M..N	Entre M y N veces, siendo M y N números concretos

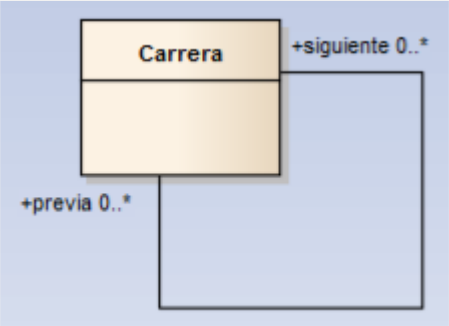
Algunos ejemplos con multiplicidad:



3.4 Asociar una clase a sí misma

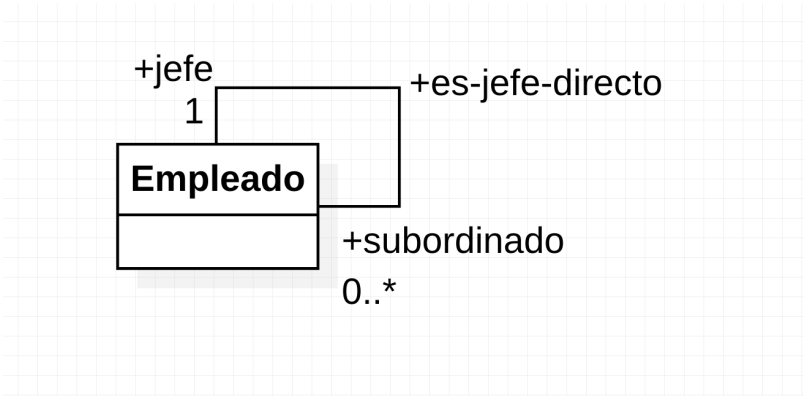
Cuando encontramos una misma clase en los dos extremos de una asociación, hablamos de **asociaciones reflexivas**, que unen entre sí instancias de una misma clase.

En estos casos sí que es conveniente añadir un nombre o rol a cada uno de los extremos de la asociación, sobre todo si la multiplicidad en ambos extremos no es la misma.

> Ejemplo: para poder participar en una `Carrera`, es necesario haber participado en otras carreras previas.
>
> 
>

Las asociaciones reflexivas sirven principalmente para describir, dentro del conjunto de instancias de una clase:

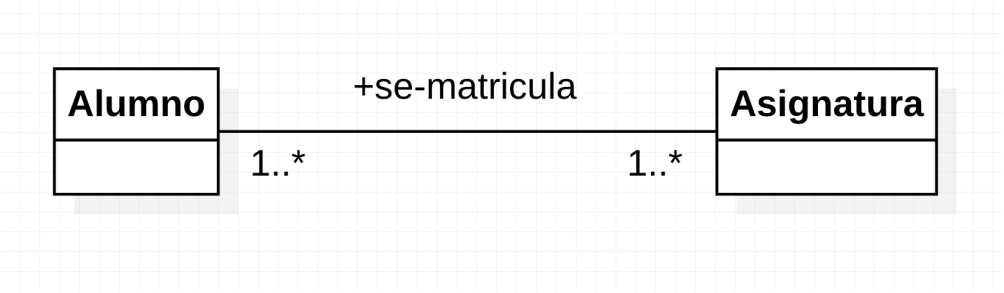
- Grupos de instancias
- **Una jerarquía dentro de las instancias (una relación de orden)**.



3.5 Las clases-asociaciones o clases de asociación.

Hay algunas asociaciones que pueden incorporar sus propios atributos. El valor de estos atributos es específico al vínculo entre las dos instancias que se están relacionando mediante la asociación.

Posiblemente, seamos capaces de comprenderlo mejor mediante un ejemplo:

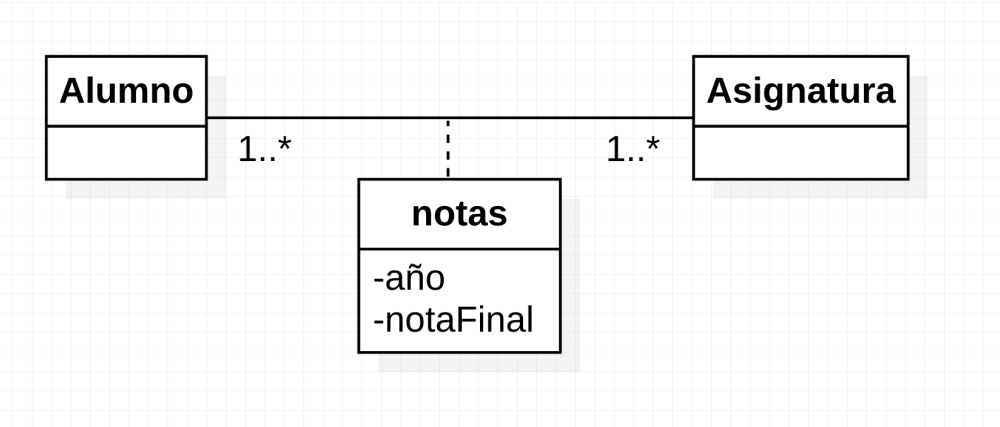


En este caso, la asociación `se-matricula` podría incorporar también las notas finales del alumno en cada asignatura. Pero claro:

- Las notas no son del alumno, ya que un alumno puede tener notas diferentes en asignaturas diferentes (y en años diferentes).
- Las notas tampoco son de las asignaturas, ya que cada asignatura puede tener notas diferentes, obtenidas por alumnos diferentes.

Por tanto, las notas no son información (un atributo) ni de la instancia de `Alumno` ni de la de `Asignatura`, sino de la asociación entre ambas.

Para poder reflejar casos como este, podemos definir una *clase de asociación*. Se trata de una asociación especial, representada con el mismo rectángulo que cualquier otra clase, pero que *pende* de la asociación entre las otras dos clases, y conectada a dicha asociación mediante una línea punteada.



> Podemos ver como al añadir una clase de asociación, el nombre de la asociación desaparece. Más que desaparecer, se traslada como nombre de la clase de asociación.

3.6 Composición y Agregación

Un objeto puede ser complejo, y estar compuesto a su vez por otros objetos. La asociación que une estos objetos (componente y compuesto) es la *composición*, definida a nivel de sus clases, pero cuyos vínculos se establecen entre las instancias.

> Podemos imaginar una ordenador de sobremesa (en particular, lo que a veces se conoce como torre o CPU), que es un objeto complejo compuesto de placa base, microprocesador, memoria, tarjeta gráfica, fuente de alimentación, ...

La forma de representar la interrelación entre estos objetos es la **asociación de composición**. Este tipo de relación permite asociar un objeto complejo con los objetos que lo constituyen, es decir, sus componentes.

Existen dos formas de composición:

- ***Composición fuerte***, también conocida a secas como *composición*.
- ***Composición débil***, conocida como *agregación*.

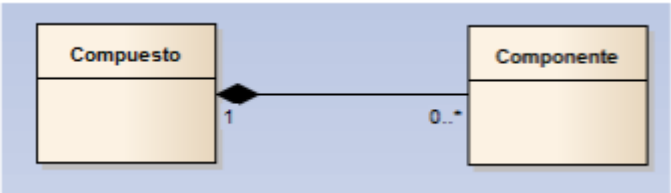
3.6.1 Composición fuerte (composición)

Es una forma de composición en la que un objeto (el TODO) está formado por varios componentes, cuya vida (ámbito, en términos de variables en un lenguaje de programación) está ligada a la del todo. De esta forma:

- Los componentes no pueden ser compartidos por varios objetos compuestos.

- La multiplicidad máxima del objeto compuesto es siempre 1.

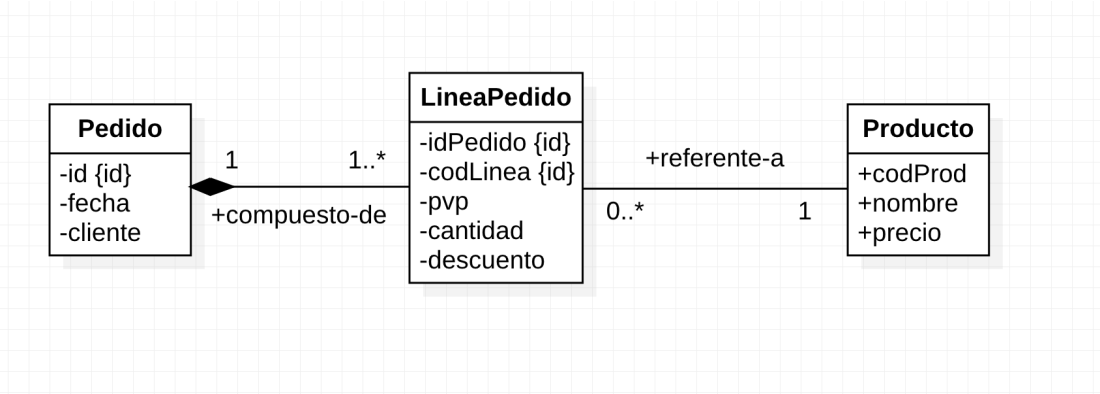
Como la vida de los componentes está ligada a la del objeto compuesto, esto implica que la desaparición del compuesto implica la desaparición de los componentes (una especie de *borrado en cascada*).



> En ocasiones, las asociaciones de composición fuerte no llevan nombre, porque se deduce que el nombre de la asociación es `compuesto-de` o `tiene`.

La asociación de composición se representa como el resto de asociaciones, agregando un rombo relleno en el extremo del compuesto.

Un ejemplo clásico de composición en cualquier sistema que implica compras o ventas es el siguiente:



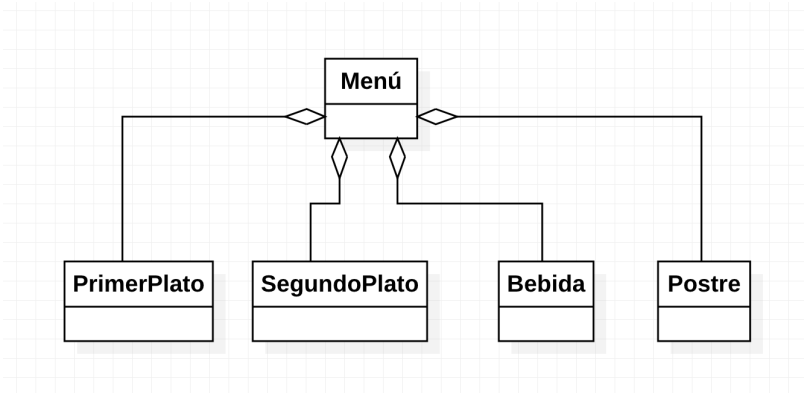
3.6.2 La composición débil (agregación)

La composición débil, conocida habitualmente como agregación, impone menos restricciones que su hermana fuerte.

En el caso de la agregación:

- Los componentes pueden ser compartidos por varios compuestos (de la misma asociación de agregación, o de varias asociaciones diferentes).
- La destrucción del compuesto no implica la destrucción de las componentes.

En este ejemplo, algo rebuscado, encontramos cuatro asociaciones de agregación:



> De alguna forma, podemos decir que la agregación es una asociación que *agrupa* un conjunto de instancias de una clase.

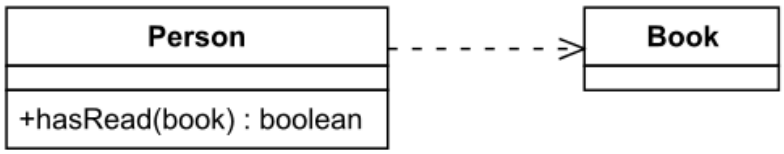
3.6.3 Diferencias entre agregación y composición

Aspecto	Agregación	Composición
Representación	Rombo transparente	Rombo negro
Varias instancias del compuesto comparten los componentes	Sí	No
Destrucción de los componentes al destruir el compuesto	No	Sí
Cardinalidad del compuesto	Cualquiera	0..1 o 1

3.7 Otras asociaciones

Los diagramas de clases UML permiten el uso de otro tipo de asociaciones entre clases, como es la **relación de dependencia**. En UML, una relación de dependencia es una relación en la que un elemento, el cliente, usa o depende de otro elemento, el proveedor. Se pueden usar relaciones de dependencia en diagramas de clases, diagramas de componentes, diagramas de implementación y diagramas de casos de uso para indicar que un cambio en el proveedor puede requerir un cambio en el cliente. También se puede usar una relación de dependencia para representar la precedencia, donde un elemento del modelo debe preceder a otro.

Normalmente, las relaciones de dependencia no tienen nombres, y se representan por una línea punteada, en lugar de una línea continua, que va desde el cliente al proveedor (la punta de flecha apunta al proveedor).



Como esta relación de dependencia puede representar diferentes tipos de asociaciones, lo que sí se le suele añadir es un estereotipo (entre dobles ángulos <<>>); algunos de los más habituales aparecen en la siguiente tabla:

Tipo de dependencia	Estereotipo	Descripción

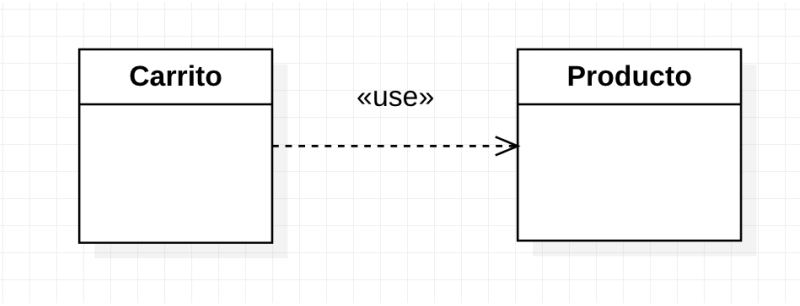
Tipo de dependencia	Estereotipo	Descripción
Abstracción	`<<abstraction>>`	Indica que dos clases del modelo representan el mismo concepto en diferentes niveles de abstracción, o desde diferentes puntos de vista.
Binding	`<<bind>>`	Conecta argumentos de plantilla a parámetros de plantilla para crear elementos de modelo a partir de plantillas.
Realización	`<<realize>>`	Indica que el cliente es una implementación del proveedor (como vimos con las interfaces).
Sustitución	`<<substitute>>`	Indica que el elemento del modelo de cliente toma el lugar del proveedor; el elemento del modelo del cliente debe ajustarse al contrato o interfaz que establece el elemento del modelo del proveedor.
Uso	`<<use>>`, `<<call>>`, `<<create>>`, `<<instantiate>>` o `<<send>>`	Indica que un elemento del modelo requiere otro elemento del modelo para su implementación.

De entre todas, la dependencia de **uso** es la más frecuente.

> Para saber más sobre `<<abstraction>>` puedes echar un vistazo aquí

> Para saber más sobre `<<bind>>` puedes echar un vistazo aquí.

Un ejemplo sería, en una aplicación de comercio electrónico, las clases `Carrito`, donde se almacenan los elementos que se quieren comprar, y la clase `Producto`. A nivel estructural no tienen ninguna asociación, pero la clase `Carrito` usará instancias de la clase `Producto` en alguno de sus métodos, como `add(p : Producto)`, y es posible que tenga un atributo de tipo `Map<Producto, Integer>`.



4. Diferencias entre un diagrama de clases en la fase de análisis y otro en la fase de diseño

Como ya indicábamos en uno de los apartados anteriores, todo lo que hemos trabajado ahora es de aplicación, principalmente, cuando estamos realizando un diagrama de clases en la fase de análisis. Pero, ¿para qué sirve un diagrama en dicha fase del desarrollo de software?

> **¿Qué es análisis y diseño?**
> El Análisis pone énfasis en una investigación del problema y los requisitos, en vez de ponerlo en una solución. Por ejemplo, si se desea un nuevo sistema de información informatizado para una biblioteca, ¿cómo se utilizará?
> **"Análisis"** es un término amplio, es más adecuado calificarlo, como análisis de requisitos (un estudio de los requisitos) o análisis de objetos (un estudio de los objetos del dominio).
> El **Diseño** pene énfasis en una solución conceptual que satisface los requisitos, en vez de ponerlo en la implementación. Por ejemplo, una descripción del esquema de una base de datos y objetos software. Finalmente, los diseños pueden ser implementados. Como con el análisis, es más apropiado calificar el término como diseño de objetos o diseño de bases de datos
> El análisis y el diseño se han resumido en la frase **hacer lo correcto (análisis)**, y **hacerlo correcto (diseño)**.

4.1 Diagramas de clases en el análisis: definición de un modelo de dominio

La finalidad del análisis orientado a objetos es crear una descripción del dominio desde la perspectiva de la clasificación de objetos. Una descomposición del dominio conlleva una identificación de los conceptos, atributos y asociaciones que se consideran significativas. El resultado se puede expresar en un modelo del dominio, que se ilustra mediante un conjunto de diagramas que muestran los objetos o conceptos del domino. Uno de los principales diagramas utilizados para ello es el diagrama de clases.

4.2 Diagramas de clases en el diseño

La finalidad del diseño orientado a objetos es definir los objetos software y sus colaboraciones. Una notación habitual para ilustrar estas colaboraciones es el diagrama de interacción. Muestra el flujo de mensajes entre los objetos software y, por tanto, la invocación de métodos (ver PDF de introducción.)

> Los diseños de los objetos software y los programas se inspiran en los dominios del mundo real (identificados en la fase del análisis), pero no son modelos directos o simulaciones del mundo real.

Además de la vista dinámica de las colaboraciones entre los objetos que se muestra mediante los diagramas de interacción, es útil crear una vista estática de las definiciones de las clases mediante un diagrama de clases de diseño

A diferencia del modelo de dominio, este diagrama no muestra conceptos del mundo real, sino clases software.

4.3 Principales diferencias entre los diagramas de clases del análisis y del diseño

Aspecto	Análisis	Diseño
Navegación	En la mayoría de las ocasiones no se especifica la dirección de navegación.	En la mayoría de las ocasiones sí se especifica, ya que denota el sentido del paso de mensajes entre objetos.

Aspecto	Análisis	Diseño
Tipos de datos	No se suelen indicar, solamente en casos muy específicos.	Se suele añadir a todos los atributos. Es posible que se utilicen, en lugar de los tipos básicos UML, tipos del lenguaje de programación a utilizar en la fase de codificación.
Asociaciones y atributos	La asociación se representa únicamente como la línea que conecta a las clases, pero no añade atributos a las clases.	La asociación, además de representarse como en el análisis, tendrá un impacto en cada clase que conecte, añadiendo como atributo una referencia (o una colección de referencias) del tipo de la clase opuesta.
Métodos	Rara vez se añaden métodos en esta fase, y si se hace, suele indicarse solamente su nombre.	Se añaden todos los métodos necesarios, siendo específicos (si es posible) en tipo de retorno, número de argumentos y tipo.

Podríamos ver las diferencias en los siguientes diagramas: el primero (en la fase de análisis) y el segundo (en la fase de diseño, más orientado a ser elemento de entrada para la implementación):

Diagrama de clases de la fase de Análisis

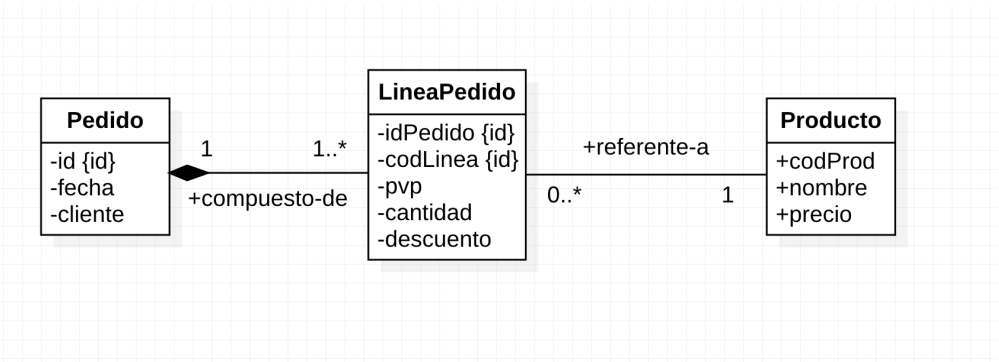
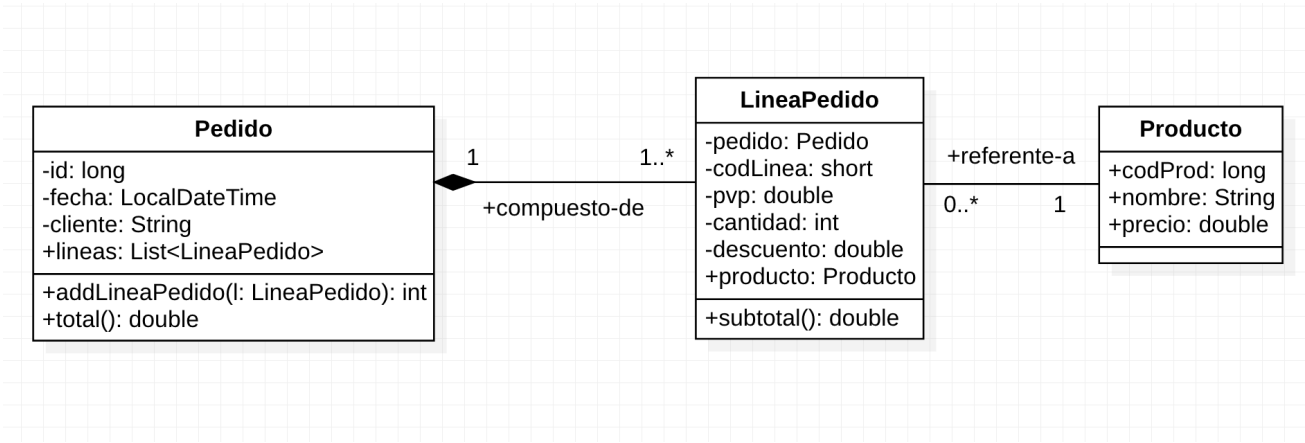


Diagrama de clases de la fase de Diseño



6. Pistas para realizar un diagrama de clases

2ª PARTE. CASOS DE USO

7. Requisitos o requerimientos

Los requerimientos son una descripción de las necesidades o deseos de un producto. La meta primaria de la fase de requerimientos es identificar y documentar lo que en realidad se necesita, en una forma que claramente se lo comunique al cliente y a los miembros del equipo de desarrollo. El reto consiste en definirlos de manera inequívoca, de modo que se detecten los riesgos y no se presenten sorpresas al momento de entregar el producto.

En esta fase de requisitos, se suelen documentar las *funciones del sistema* y los *atributos del sistema*.

Las funciones del sistema son lo que éste habrá de hacer, por ejemplo autorizar los pagos a crédito. Hay que identificarlas y listadas en grupos cohesivos y lógicos.

> Con el objeto de verificar que algún X es de verdad una función del sistema, la siguiente oración deberá tener sentido: **El sistema deberá hacer <X>.** Por ejemplo: El sistema deberá autorizar los pagos a crédito.

En cambio, los **atributos del sistema** son cualidades no funcionales -entre ellas la facilidad de uso- que a menudo se confunden con las funciones. Nótese que "facilidad de uso" no encaja en la oración de verificación: *El sistema deberá hacer la facilidad de uso*. Los atributos no deben formar parte del documento de las especificaciones funcionales del sistema, sino de un documento independiente que especifica sus atributos (conocidos también como *Requisitos No Funcionales*).

Las funciones, en ocasiones, se organizan en categorías, que podrían ser:

- Evidente: Debe realizarse, y el usuario debería saber que se ha realizado.
- Oculta: Debe realizarse, aunque no es visible para los usuarios (por ejemplo, almacenar información en un mecanismo de persistencia).
- Superflua: Opcionales; su inclusión no repercute significativamente en el costo ni en otras funciones.

Si el sistema que estamos modelando es el de un PDV (Punto de Venta), las funcionalidades podrían ser las siguientes:

FUNCIONES BÁSICAS

- R1.1: Registra la venta en proceso (actual): los productos comprados (Evidente).
- R1.2: Calcula el total de la venta actual; se incluyen el impuesto y los descuentos (Evidente).
- R1.3: Captura la información sobre el objeto comprado usando su código de barras y un lector o usando una captura manual de un código del producto; por ejemplo, un código universal de producto (UPC) (Evidente).
- R1.4: Reduce las cantidades del inventario cuando se realiza una venta (Oculta).
- R1.5: Se registran las ventas efectuadas (Oculta).
- R1.6: El cajero debe introducir una identificación y una contraseña para poder utilizar el sistema (Evidente).
- R1.7: Ofrece un mecanismo de almacenamiento persistente (Oculta).
- R1.8: Muestra la descripción y el precio del producto registrado. (Evidente).

FUNCIONES DE PAGO

- R2.1: Maneja los pagos en efectivo, capturando la cantidad ofrecida y calculando el saldo deudor (Evidente).
- R2.2 Maneja los pagos a crédito, capturando la información crediticia a partir de una lectora de tarjetas o mediante captura manual, y autorizando los pagos con el servicio de autorización (externa) de créditos de la tienda a través de una conexión por internet (Evidente).
- R2.3 Registra los pagos en el sistema de cuentas por cobrar, pues el servicio de autorización de crédito debe a la tienda el importe del pago (Oculta).

Los requisitos funcionales o atributos del sistema podrían ser:

- Facilidad de uso
- Tolerancia a fallos
- Tiempo de respuesta
- ...

Estos requisitos se pueden concretar dando restricciones de frontera, algo así como condiciones que se deben de cumplir. Por ejemplo:

- Tiempo de respuesta: cuando se registre un producto vendido, la descripción y el precio aparecerán en menos de 2 segundos.
- Tolerancia a fallos: debe registrar los pagos a crédito autorizados que se hagan a las cuentas por cobrar en un plazo de 24, aun cuando se produzcan fallos de energía o en el servidor.
- Facilidad de uso: se debe maximizar una navegación fácil con teclado y que no sea necesario el uso de ratón.

8. Casos de uso

Una técnica excelente que permite mejorar la comprensión de los requerimientos es la **creación de casos de uso**, es decir, descripciones narrativas de los procesos del dominio. UML incluye formalmente el concepto de casos de uso y sus diagramas de uso.

> **Un caso de uso es un documento narrativo que describe la secuencia de eventos de un actor (agente externo) que utiliza un sistema para completar un proceso**. Los casos de uso son historias o casos de utilización de un sistema; no son exactamente los requisitos ni las especificaciones funcionales, sino que ejemplifican e incluyen tácitamente los requerimientos en las historias que narran.

Si lo que queremos es representar un caso de uso en un diagrama, tenemos el siguiente icono:

A continuación, podemos ver en qué consistiría un caso de uso de alto nivel: **comprar productos**.

- Caso de uso: comprar productos

- Actores: cliente, cajero
- Tipo: Primario (se explicará luego)
- Descripción: Un Cliente llega a la caja registradora con los artículos que comprará. El Cajero registra los artículos y cobra el importe. Al terminar la operación, el Cliente se marcha con los productos.

Los apartados descritos son una pista, ya que UML no especifica un formato rígido, y puede modificarse para atender las necesidades de un proyecto concreto.

> Un caso de uso de alto nivel describe un proceso de negocio o dominio muy brevemente. Se utiliza en las primeras fases del análisis para entender rápidamente la funcionalidad del sistema. Conviene comenzar con los casos de uso de alto nivel para lograr rápidamente entender los principales procesos globales.

Un caso de uso expandido muestra más detalles que uno de alto nivel; este tipo de casos suelen ser útiles para alcanzar un conocimiento más profundo de los procesos y de los requerimientos.

El mismo caso de uso anterior **comprar productos**, en formato expandido, podría tener la siguiente estructura:

SEGUIR POR LA PÁGINA 37 DE LARMAN, PERO USANDO LA PLANTILLA DE CLASSROOM PARA CASOS DE USO