



*Plan 2024*

# *Programación II*

*Parte II - v.1*

*Ing. David CECCHI*

*UTN*



## UML

Un modelo es una representación simplificada de la realidad, capta los aspectos más relevantes (desde un punto de vista determinado), y simplifica u omite el resto. Permite abstraer la realidad a un nivel comprensible, sin perderse en los detalles.

El modelo de un sistema de software ayuda a los desarrolladores a explorar fácilmente diversas arquitecturas y soluciones de diseño antes de proceder con la codificación.

UML (Lenguaje de Modelado Unificado) es un lenguaje de modelado de propósito general que pueden utilizar todos los modeladores; es extenso y variado, pensado para ser utilizado a diferentes niveles y en múltiples etapas del ciclo de vida de desarrollo. No tiene propietario y está basado en el común acuerdo de gran parte de la comunidad informática.

A través de UML pueden representarse diferentes modelos, entre ellos: diagrama de clases, diagrama de objetos, diagrama de casos de uso, diagrama de transición de estados, etc.

## DIAGRAMA DE CLASES

En un diagrama de clases, como su nombre lo indica, se representan las clases y sus relaciones. Se pueden incluir interfaces y tipos de enumeración.

Cada clase se representa por un recuadro constituido por tres segmentos: en el segmento superior se detalla el nombre de la clase, en el segmento medio los atributos y en el último segmento los métodos.

Adicionalmente, sobre cada propiedad y método se podrá indicar el grado de visibilidad y el tipo de dato asociado. A su vez, cada tipo de relación cuenta con una representación predeterminada, permitiendo además indicar la cardinalidad de la misma, si corresponde.

El nivel de detalle o abstracción representado dependerá de la etapa de modelado en que se encuentre el proceso y del objetivo que se pretenda alcanzar con el diagrama en ese momento. Es decir, si se está en una fase inicial del proceso de modelado quizás con solo mencionar los nombres de las clases y sus vinculaciones sin profundizar en los detalles de las mismas se esté cumpliendo el objetivo. Luego, en etapas posteriores, donde se requiera especificidad para avanzar, se profundizará en los detalles pendientes dando lugar a un diagrama completo.

Si es necesario extender la aplicación del lenguaje a través de algún elemento para el cual no existe una representación predeterminada, es posible recurrir a los estereotipos, los mismos se definen encerrando su nombre o denominación entre símbolos dobles de menor y mayor, ej: <<nombre del estereotipo>>.

Visibilidades más relevantes		
Símbolo	Definición	Alcance
-	Privada	El atributo o método puede ser accedido solo dentro de la misma clase
#	Protegida	El atributo o método puede ser accedido dentro de la misma clase y desde las clases que hereden de ella.
+	Pública	El atributo o método puede ser accedido desde cualquier lugar de la aplicación.

*Los atributos estáticos (atributos de clase) no cuentan con un modificador de visibilidad, solo se notan subrayados.*

Los **atributos** se notan con el siguiente formato:

*visibilidad nombre\_atributo: tipo = valor\_inicial*

Aunque el formato detallado anteriormente es el estándar, es común simplificar el mismo especificando el nombre y el tipo o únicamente el nombre.

Los **métodos** se notan con el siguiente formato:

*visibilidad nombre\_metodo (parametros): tipo\_devuelto*

Aunque el formato detallado anteriormente es el estándar, es común simplificar el mismo especificando el nombre del método y, en ocasiones, el tipo devuelto.

A los métodos que no retornan un valor, se suele no especificar el tipo de dato “devuelto” (void).

El marcador de visibilidad se puede suprimir. La ausencia de un marcador de visibilidad solo indica que la visibilidad no se muestra, no implica que sea pública o no esté definida.

A continuación, se presentan tres alternativas válidas para la representación de la clase Empleado:




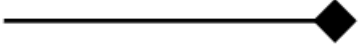
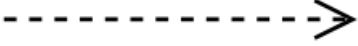
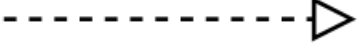

Empleado	Empleado	Empleado
-legajo: int -nombre: string -fechaNac: date -edad: int {readOnly}	-legajo: int -nombre: string -fechaNac: date -edad: int {readOnly}	legajo: int nombre: string fechaNac: date edad: int {readOnly}
+Empleado() +Empleado(int, string, date) +getLegajo() +setLegajo() +getNombre() +setNombre() +getFechaNac() +setFechaNac() +getEdad()	+Empleado() +Empleado(int, string, date)	Empleado() Empleado(leg: int, nom: string, fec: date)

Se debe tener presente que en POO los atributos no suelen definirse como públicos, de hacerlo se estaría brindando acceso directo a los mismos desde el exterior y consecuentemente no se respetaría el concepto de encapsulamiento. Si se requiere que un atributo sea accesible desde el exterior, el mismo se debe mantener como privado e implementar su método de propiedad, separando getter y setter, con visibilidad pública.

En las dos últimas representaciones, con el objetivo de simplificar la legibilidad del diagrama, se asume implícitamente que todas las propiedades cuentan con getter y setter, exceptuando Edad para la cual se ha especificado el modificador readOnly. Dicho modificador indica que para Edad únicamente se deberá definir el getter (solo lectura).

Es posible especificar si un atributo debe aceptar múltiples valores, es decir, representará una colección. Para ello, se deben utilizar corchetes indicando en su interior la cantidad de instancias aceptadas, ej: dirección: String [\*].

## TIPOS DE RELACIONES ENTRE CLASES

Representación	Tipo de Relación
	Asociación Bidireccional
	Asociación Unidireccional
	Agregación
	Composición
	Dependencia
	Realización/Implementación
	Generalización/Herencia

### RELACIÓN DE ASOCIACIÓN

Una relación de asociación describe una vinculación entre objetos, estos objetos pueden pertenecer a clases diferentes o a la misma clase (autoasociación).

#### Dirección

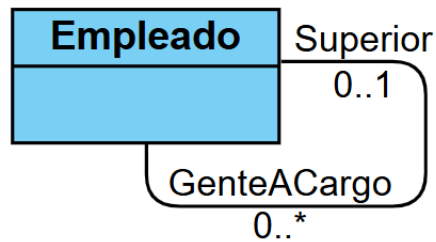
Una asociación puede ser bidireccional o unidireccional. Bidireccional implica que será posible navegar desde instancias de la clase A hacia instancias de la clase B, y desde instancias de la clase B será posible acceder a instancias de la clase A. En una relación unidireccional se incorpora una flecha en uno de los extremos que indicará el único sentido de navegación.

#### Cardinalidad

En una asociación, es necesario indicar cuántos objetos de cada clase pueden estar involucrados en la relación. En otras palabras, es necesario indicar la cardinalidad (o multiplicidad) de la asociación. Dadas dos clases A y B, la cardinalidad indica con cuántas instancias de A se puede relacionar cada instancia de B (valor mínimo y máximo) y viceversa. Los casos más generales de cardinalidad son los siguientes:

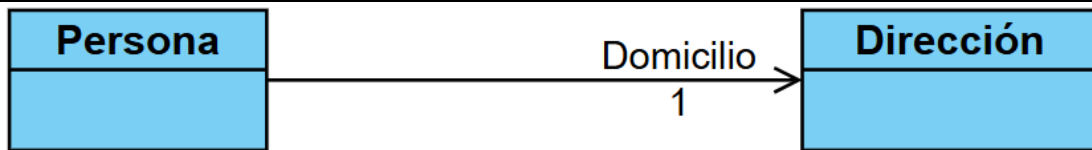
- Cardinalidad “uno a uno”: una instancia de la clase A se relaciona con una única instancia de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad “uno a muchos”: una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad “muchos a muchos”: una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con varias instancias de la clase A.

Cardinalidad	Descripción
1	Solo uno
*	Mínimo ninguno, máximo sin límite
0..*	Mínimo ninguno, máximo sin límite
1..*	Mínimo uno, máximo sin límite
N	Número exacto
N..M	Mínimo N, máximo M



Un Empleado podrá tener a cargo varios Empleados, uno o ninguno, a su vez todos tendrán un Empleado responsable de ellos o ninguno (ninguno: el propietario de la empresa figuraría sin un superior).

Las propiedades que se agregan a Empleado a partir de la relación se denominan GenteACargo y Superior.



Una Persona tendrá asociada una Dirección, mientras que una Dirección no tendrá asociada Personas. Es decir, una Dirección no conoce la Persona o Personas que la tienen asociada.

La propiedad en Persona se denomina Domicilio.

A modo de ejemplo, se podría interpretar que una Dirección contendría propiedades tales como: calle, numeración, piso, depto., etc.



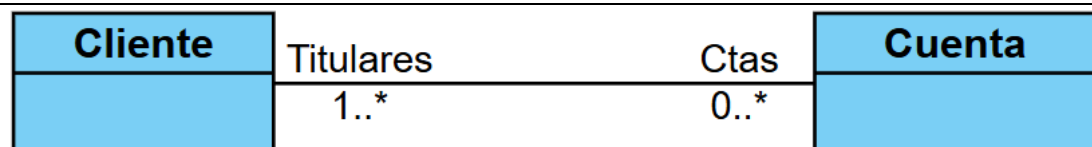
Un Cliente puede o no tener asociada una Cuenta, mientras que una Cuenta no tendrá asociado Clientes. Es decir, una Cuenta no conoce al Cliente o Clientes que la tienen asociada.

La propiedad en Cliente se denomina Cta.



Un Cliente puede o no tener asociada una Cuenta, mientras que una Cuenta siempre tendrá asociado un Cliente.

La propiedad en Cliente se denomina Cta, mientras que la propiedad en Cuenta se denomina Titular.



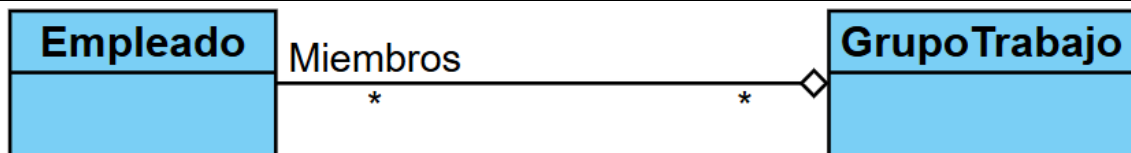
Un Cliente puede tener asociada una, varias o ninguna Cuenta, mientras que una Cuenta siempre tendrá asociado uno o más Clientes.

La propiedad en Cliente se denomina Ctas, mientras que la propiedad en Cuenta se denomina Titulares.

## RELACIÓN DE AGREGACIÓN

Es un caso especial de asociación donde se pretende aclarar que una de las clases involucradas representa un TODO y la otra una PARTE de ella, donde la parte puede existir independientemente del todo. La parte se generará fuera del Todo, y es pasada a él a través de algún medio (constructor/método).

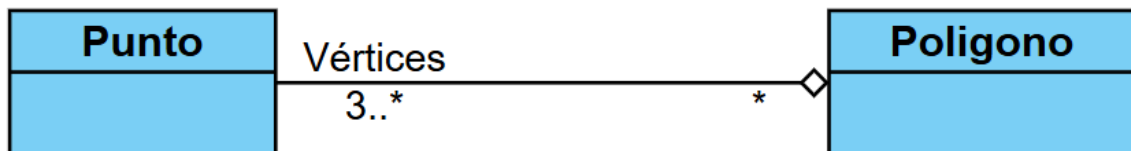
La Parte tiene existencia fuera del Todo, tiene sentido por sí misma. Incluso, un mismo objeto Parte podría ser “parte” de varios objetos Todo.



Un Grupo de Trabajo puede estar constituido por varios Empleados, y un Empleado puede formar parte de varios Grupos de Trabajo.

Eliminar un Grupo de Trabajo, no implica eliminar los Empleados.

La clase GrupoTrabajo incluiría un método que le permitiría agregar un Empleado a su colección.



Un Polígono estará constituido por 3 o más Puntos, y un Punto puede formar parte de varios Polígonos.

Eliminar un polígono, no implica eliminar los Puntos.

La clase Polígono incluiría un método que le permitiría agregar un Punto a su colección.

## RELACIÓN DE COMPOSICIÓN

Es un caso especial de asociación donde se pretende aclarar que una de las clases involucradas representa un TODO y la otra una PARTE de ella, donde la Parte no puede existir independientemente del Todo. La Parte se genera dentro del Todo, no puede ser pasada a él desde el exterior.

La Parte, no tiene de existencia fuera del Todo, no tiene sentido por sí misma. Por lo tanto, nunca un mismo objeto Parte podría ser “parte” de varios objetos Todo.

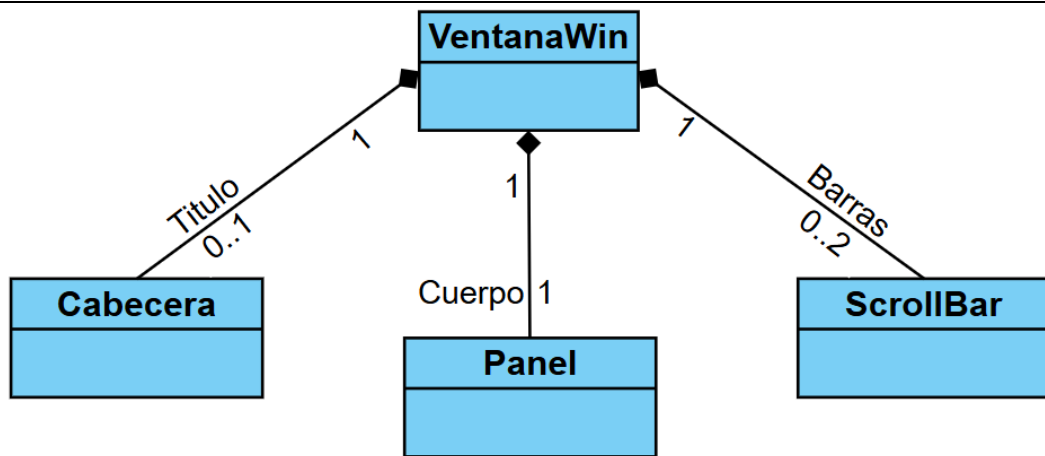
En una relación de composición hay una regla de “no compartir”.



Un Libro estará constituido por varias Hojas, y una Hoja no puede formar parte de otros Libros.

Eliminar un Libro, implica la destrucción de sus Hojas.

La clase Libro incluiría un método que le permitiría generar sus Hojas.



Una Ventana (formulario en aplicación de escritorio) estará constituida por un Panel, hasta dos scrollbars (barras de desplazamiento) y puede o no contener una Cabecera (titulo). Donde su título, cuerpo y barras no pueden formar parte de otras Ventanas.

Eliminar una Ventana, implica la destrucción de los objetos contenidos (Cabecera, Panel y ScrollBars).

La clase VentanaWin incluiría métodos que le permitirían generar su Título, Cuerpo y Barras.

La clase Cabecera podría definir propiedades como: texto contenido, tipo de fuente, tamaño de fuente, etc.

La clase Panel podría definir propiedades como: alto, ancho, color de fondo, etc.

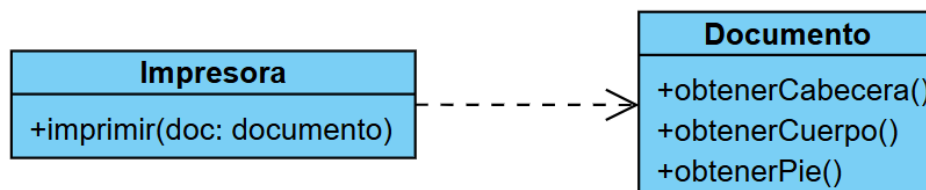
La clase ScrollBar podría definir propiedades como: orientación, posición actual, etc.

Característica	Agregación	Composición
Dos o más compuestos pueden compartir un mismo componente	Sí	No
Al destruir el compuesto se destruyen los componentes	No	Sí
Cardinalidad del compuesto	Cualquier cardinalidad	1
Representación	Rombo vacío	Rombo relleno

## RELACIÓN DE DEPENDENCIA

Una dependencia entre clases denota una relación de uso de una sobre otra. Esta situación se presenta cuando un método de la clase A recibe como parámetro un objeto de la clase B o dentro de algún método de la clase A se instancia un objeto de la clase B para desempeñar una tarea determinada. En estas situaciones, se dice que A depende de B, es decir, hace uso de sus servicios.

Por lo cual, en caso de sufrir modificaciones la clase de la cual se depende, es posible que deban introducirse cambios en la clase dependiente.



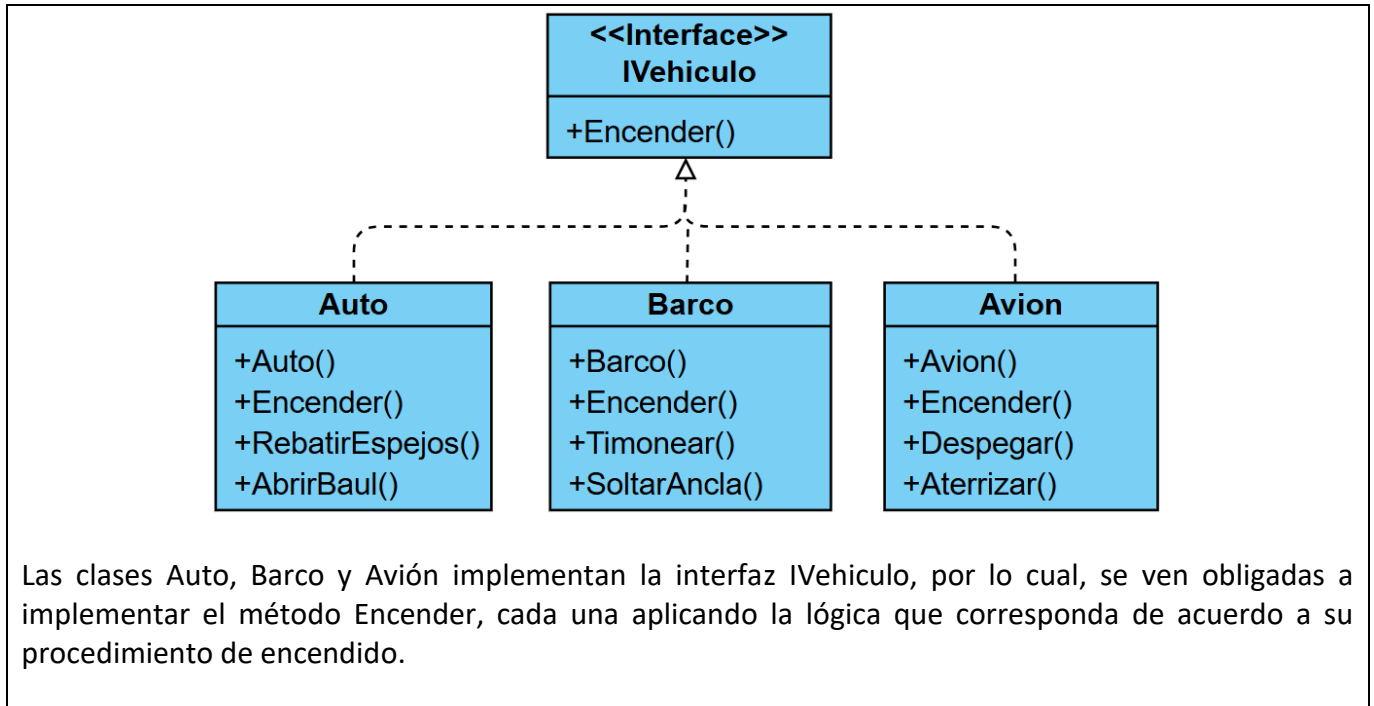
La clase Impresora, para imprimir un documento, invocará a las tres operaciones definidas en la clase Documento. En consecuencia, si los métodos de la clase Documento son eliminados, se modifican sus parámetros o el tipo de dato retornado, la clase Impresora se verá obligada a modificar la

implementación de Imprimir.

## RELACIÓN DE REALIZACIÓN/IMPLEMENTACIÓN

Una realización denota una relación entre una interfaz y una clase, donde la clase se encuentra obligada a cumplir con el contrato establecido, es decir, deberá implementar el conjunto de propiedades y métodos declarados en la interfaz.

Una interfaz se identifica con el estereotipo <<Interface>>.



## GENERALIZACIÓN/HERENCIA

La generalización es una relación entre una clase más general (denominada superclase o clase padre) y una clase más específica (denominada subclase o clase hija).

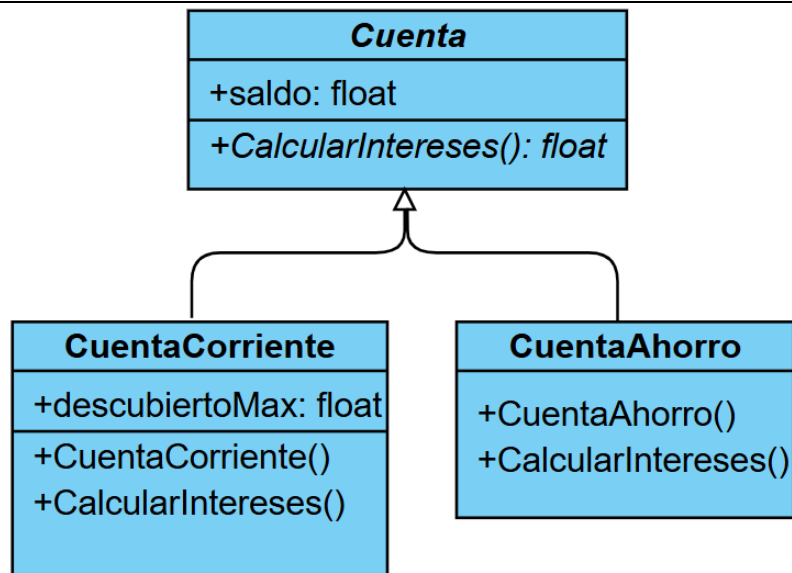
Toda subclase hereda los atributos y métodos de la superclase, y a su vez puede añadir sus propios atributos y métodos.

Las subclases representan un tipo específico de la superclase, por ello, toda instancia de la subclase puede interpretarse a su vez como instancia de la superclase. Ejemplificando, si se cuenta con la superclase Persona y una subclase de ella es la clase Cliente, implica que una instancia de la clase Cliente (más allá de ser un Cliente) no deja de ser una Persona.

Este razonamiento, permite concluir/entender que toda instancia de la subclase puede emplearse en cualquier espacio donde se requiera una instancia de la superclase.

Para indicar que una clase es abstracta su nombre se nota en cursiva, también está permitido agregar el estereotipo <<abstract>> por encima de su nombre. Los elementos virtuales y abstractos, se notan en cursiva para diferenciarlos de los concretos.





La clase Cuenta es abstracta (se encuentra en cursiva) al igual que su método `CalcularIntereses`, por lo cual, ambas subclases están obligadas a redefinirlo.