

UML y UP: Análisis y Diseño Orientado a Objetos

Módulo 1

El Diagrama de Clases (*Class Diagram*)

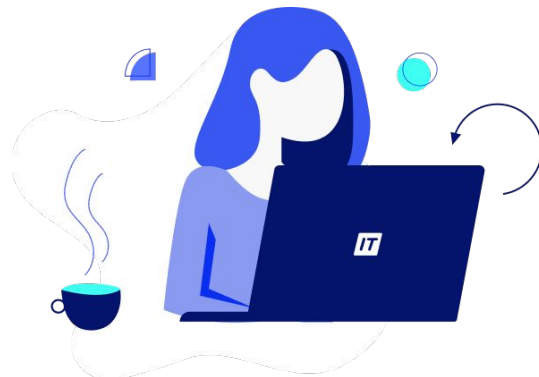
El Diagrama de Clases (*Class Diagram*)

Definición

El diagrama de clases permite **modelar la estructura del sistema desde un punto de vista estático**, mediante la conformación de las clases desde distintos enfoques de acuerdo a la etapa del proyecto. **Describe tipos de jerarquías, relaciones y abstracciones.**

Objetivo

Describir las clases del dominio, sus atributos y relaciones.



Elementos

Clase

Una clase **representa a una agrupación de cosas, es una plantilla para armar un objeto.**

En la mayoría de los casos, las clases se detectan como sustantivos en singular.

Un ejemplo puede ser la clase Auto, que es una clase representante de todos los autos posibles (autos de carrera, autos urbanos, cualquier marca, patente, color, etc.). También la clase Animal, que representa a todos los animales posibles (mamíferos, herbívoros u otros, cualquier cantidad de patas, color, etc.).

Las clases **están formadas por atributos y métodos**, y por convención, la primera letra debe estar en mayúscula.

Veamos la siguiente slide.



Qué son los atributos

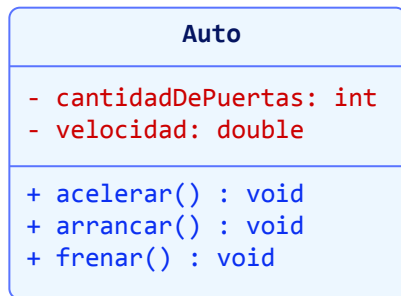
Los atributos son **características** que posee una clase y **determinan el estado que posteriormente tendrá un objeto**.

En el caso de la clase Auto, atributos pueden ser color, marca, modelo, cantidad de puertas y velocidad. Por convención, la primera letra debe estar en minúscula.

Qué son los métodos

Los métodos son las **responsabilidades (o comportamiento) que realiza una clase**. Generalmente los métodos son verbos. Por convención, la primera letra debe estar en minúscula.

Representación gráfica de una clase



Qué es el alcance o ámbito de una clase

El alcance (scope) en una clase implica la **accesibilidad o visibilidad de un método o atributo de la misma**.

Representa desde qué partes se puede acceder a dicho elemento y está determinado por los modificadores de visibilidad.

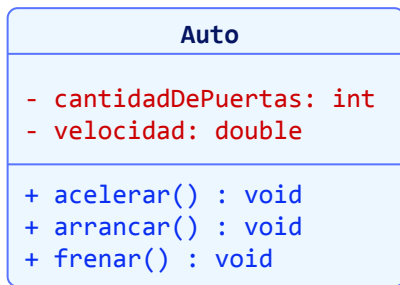


Modificadores de visibilidad y su significado

Símbolo	Scope	Visibilidad	Java
-	Privado	Sólo se tiene acceso desde la clase.	Private
+	Público	Se puede acceder desde toda la aplicación.	Public
#	Paquete	Se puede acceder desde cualquier clase del mismo paquete.	-
~	Protegido	Acceso cuando hay relación de herencia.	Protected

Ejemplo

En el ejemplo del auto visto previamente:



Vemos que los atributos cantidadDePuertas y velocidad son privados de la clase; sólo se pueden acceder desde la clase Auto.

Los métodos acelerar, arrancar y frenar son públicos, se puede acceder a ellos desde cualquier clase de la aplicación.

Hacer que un atributo sea **privado** es útil para tratar con datos o valores que **no queremos que sean accedidos más que por la clase que lo contiene**.

Las operaciones privadas tienen sentido cuando en una clase tenemos procesamiento interno y **no queremos que otras clases accedan a las operaciones** por diferentes razones: puede ser porque no tiene sentido que la accedan o porque podría resultar peligroso para su integridad.

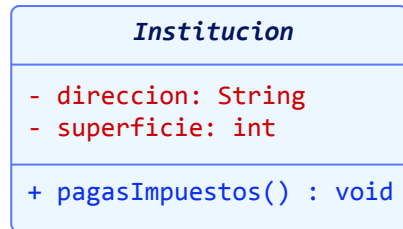
En resumen, **la visibilidad determina el alcance (scope) del atributo u operación en cuestión**.

Las clases abstractas

Estas son clases que **representan un concepto abstracto de carácter muy general**. Por ejemplo: una institución, que tiene los atributos **dirección** y **superficie**, pero no es posible determinar qué tipo de institución es.

Se trata de una clase que no se puede instanciar. Es decir que **no se puede crear un objeto a partir de esta clase**. Se utiliza como base para otras clases en la relación de generalización, pero no tiene sentido por sí sola. Las clases que no son abstractas se denominan concretas. Por convención, la primera letra debe estar en mayúscula, y la palabra en negrita e itálica.

Representación gráfica de una clase abstracta



Fuente: [Gestión del Software II](#)

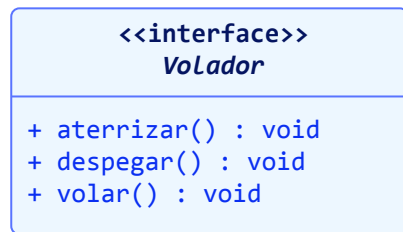
Interfaz

A diferencia de la clase, **la interfaz define únicamente un comportamiento, es decir, un conjunto de métodos que no poseen implementación.**

Las acciones de estos métodos deberán ser establecidas por las clases que decidan implementar la interfaz. Representan un “contrato” que una clase debe respetar en caso de implementar la interfaz.

Por ejemplo, se puede crear un interfaz denominada *Volador*, que tiene los métodos *despegar*, *aterrizar* y *volar*.

Por convención, para definir el nombre de una interfaz se aplican las mismas reglas que para una clase.



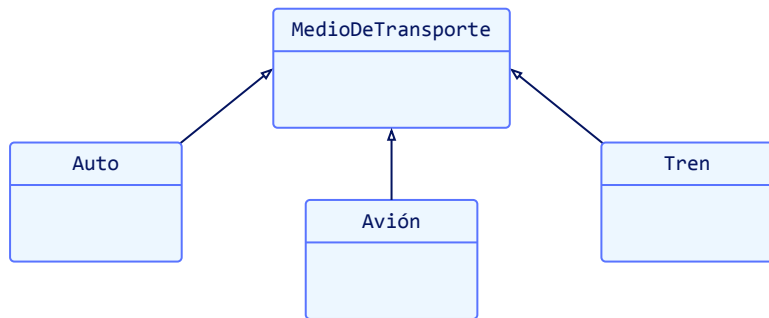
Relaciones

Generalización

La relación de generalización se produce entre dos clases. La **clase superior** es una **generalización de la clase inferior**, como así también la **clase inferior** es una **particularización de la clase superior**. A la clase superior se la denomina **súper clase**, y a la inferior, **subclase**. La subclase deberá respetar la relación “es un” o “*is a*”, que representa a la relación de generalización.

Al construir varias relaciones de este tipo entre clases, se genera la jerarquía de clases (o árbol de clases).

En términos de un lenguaje de programación, es posible entenderla como herencia. Por ejemplo, la clase `MedioDeTransporte` puede ser una super-clase, y subclases pueden ser `Auto`, `Avión` y `Tren`.



Asociación

La relación de asociación **representa una asociación entre dos clases**, donde una clase “es socia” de otra o tienen algún tipo de relación. Es común describir con un nombre definido por el usuario la relación entre ambas.

Por ejemplo, la clase Cuidador tiene una relación con la clase Perro, donde Cuidador “cuida” al Perro. Es posible determinar la multiplicidad de los extremos de la relación, en el caso anterior un cuidador cuida de uno a muchos perros.

Existen dos relaciones denominadas *Composición* y *Agregación* que son casos particulares de la relación de Asociación.

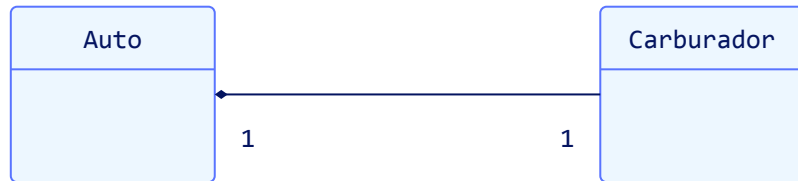


Composición

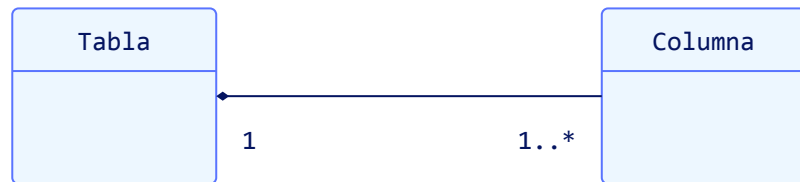
La relación de composición se puede describir como “está compuesta por”, ya que **una clase determina la existencia de la otra**. Si *Clase1* está compuesta por *Clase2* entonces *Clase1* determina la existencia de *Clase2*. *Clase2* no podrá existir por sí sola si no existe *Clase1*, es decir que *Clase1* controla el tiempo de vida de *Clase2*. Si la *Clase1* es eliminada, también será eliminada *Clase 2*, es decir que si se elimina el “todo” (*Clase1*), también serán eliminadas sus partes (*Clase2*).

UML denomina a la relación como “*strong has-a relationship*”, representando un fuerte sentido de pertenencia del “todo” hacia la “parte”.

Por ejemplo, un Auto está compuesto por un Carburador, con lo cual el Carburador no tiene sentido por sí solo si no existe el Auto. Dicho Carburador puede utilizarse únicamente para ese Auto, y no para otros, es decir que el mismo Carburador no puede utilizarse al mismo tiempo en más de un Auto



Otro ejemplo muy ilustrativo puede ser la relación entre una *Tabla* y una *Columna*, donde la *Columna* es construida si la *Tabla* es construida, y la *Columna* es eliminada si la *Tabla* es eliminada.



Agregación

La relación de agregación se puede describir cómo “tiene como partes a”. A diferencia de la composición, si *Clase1* tiene como partes a *Clase2*, *Clase1* no determina la existencia de *Clase2*. ***Clase2* puede existir aún cuando *Clase1* no exista, es decir tiene sentido por sí sola.**

Clase1 no controla el tiempo de vida de *Clase2*. Si la *Clase1* es eliminada, puede no ser eliminada *Clase2*, es decir que si se elimina el “todo” (*Clase1*), no necesariamente serán eliminadas sus partes (*Clase2*).

UML denomina a la relación como “*weak has-a relationship*”, representando un débil sentido de pertenencia del “todo” hacia la “parte”.

Por ejemplo, una *OrdenDeCompra* tiene como partes a uno o muchos *Producto(s)*, pero si el *Producto* no forma parte de ninguna *OrdenDeCompra*, sigue teniendo sentido por sí mismo. Si la *OrdenDeCompra* es eliminada, el *Producto* no será eliminado.



Implementación o Realización

La implementación se produce entre una clase y una interfaz. **La clase que implementa la interfaz tiene la obligación de implementar todos los métodos que forman parte de esa interfaz.**

Pongamos un ejemplo: un Avión para “saber volar” deberá implementar una interfaz denominada Volador, que incluye los métodos despegar, aterrizar y volar.



Multiplicidad o cardinalidad de una asociación

La multiplicidad de una asociación determina cuántos objetos de cada tipo intervienen en la relación: **el número de instancias de una clase que se relacionan con UNA instancia de la otra clase.**

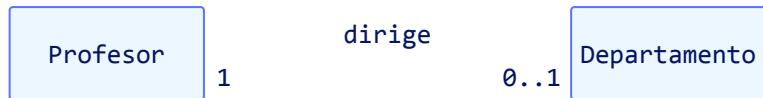
Cada asociación tiene dos multiplicidades (una para cada extremo de la relación).

Para especificar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y la multiplicidad máxima: ("Mínima" .. "Máxima")

Multiplicidad	Significado
1	Uno y sólo uno
0..1	Cero o uno
N..M	Desde N a M
*	Cero o varios
0..*	Cero o varios
1..*	Uno o varios (Al menos uno)

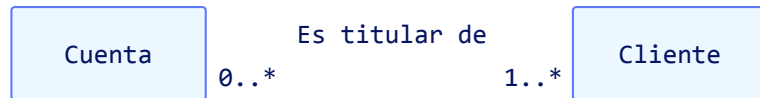
Cuando la multiplicidad mínima es 0, la relación es opcional. Una multiplicidad mínima mayor o igual que 1 establece una relación obligatoria.

Ejemplos



Todo departamento tiene un profesor que lo dirige.

Un profesor puede dirigir un departamento.



Relación opcional

Un cliente puede o no ser titular de una cuenta.

Relación obligatoria

Una cuenta ha de tener un titular como **mínimo**.



Todo profesor pertenece a un departamento.

A un departamento pueden pertenecer varios profesores.

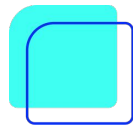
Clases estereotipadas

Qué es un estereotipo de clase

El estereotipo representa la construcción de un nuevo elemento de UML que extiende a partir de uno ya existente. En el caso de las clases, **representa a una categoría o un tipo nuevo de clases**. Representan una funcionalidad determinada que está identificada por su nombre.



El Proceso Unificado de Desarrollo (presentado más adelante) utiliza tres estereotipos de clases que se han estandarizado en el mercado, estos son Boundary, Control y Entity.



El estereotipo *Boundary*

El estereotipo Boundary se usa para **representar clases que se encuentran en el límite (*bound*) del sistema**. Estas clases representan a la interfaz de usuario dentro de un sistema, generalmente, implementadas como ventanas.

Se utilizan para capturar la interacción entre el usuario y el sistema a nivel de pantalla.

Estas clases dentro de una arquitectura multicapa generalmente pertenecen a la Capa de Presentación (PL o *Presentation Layer*).

En el patrón de diseño M-V-C el estereotipo Boundary representa a la **vista**.

El estereotipo Control

El estereotipo Control se utiliza para **representar clases que se encargan de controlar los procesos de negocios**. Son clases que llevan a cabo las reglas de negocios, realizando la coordinación entre las clases del tipo Boundary y las clases del tipo Entity. Se encargan de la organización y planificación de actividades.

Estas clases dentro de una arquitectura multicapa generalmente pertenecen a la Capa de Negocios (BL o *Business Layer*).

En el patrón de diseño M-V-C representa al **controlador**.

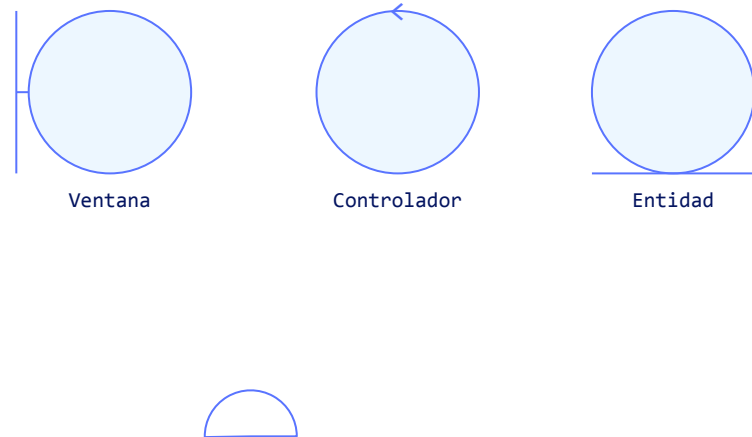
El estereotipo Entity

El estereotipo Entity se utiliza para almacenar o persistir información propia del sistema.

Estas clases dentro de una arquitectura multicapa generalmente pertenecen a la Capa de Acceso a Datos (DAL o *Data Access Layer*).

En el patrón de diseño M-V-C representa al **modelo**.


Representación gráfica



Aplicación

Modelo de Análisis o Modelo Conceptual

Uno de los posibles usos del diagrama de clases es la construcción del denominado **Modelo Conceptual**. Este modelo consiste en uno o varios diagramas de clase contruidos por un Analista Funcional que está basado en la detección de clases (junto con sus atributos y posibles métodos) y sus relaciones pero desde un punto de vista funcional, es decir, dentro de la de la etapa de Análisis.



No se definen características propias de un lenguaje de programación, sino que se intenta reflejar la realidad.

En algunos casos se categorizan las clases como *entity / controller / boundary*, que son estereotipos de RUP (*RationalUnifiedProcess*) presentados más adelante.

Adicionalmente, es posible utilizar una CRC Card (*ClassResponsibility and Collaboration*) para detallar el nombre de la clase, descripción, atributos y responsabilidades.


Modelo de Diseño

El modelo de diseño es el conjunto de diagramas de clases (puede ser uno solo) que **se utiliza como base para realizar la codificación de la aplicación**. El encargado de construirlo es el Diseñador, que debe definir todo lo que sea necesario para que el desarrollador pueda codificar sin problemas.

El diseñador toma, como uno de los documentos de entrada al Modelo de Análisis, y define nuevas clases (en general las detectadas en Análisis se mantienen) que tienen un perfil netamente de diseño y resuelven cuestiones técnicas y ya no de negocios.

A partir del Modelo de Diseño, se genera el código fuente de manera automática que será tomado como base por los desarrolladores. De este modo el programador no toma decisiones ni de análisis ni de diseño, y queda condicionado a programar en el código recibido.

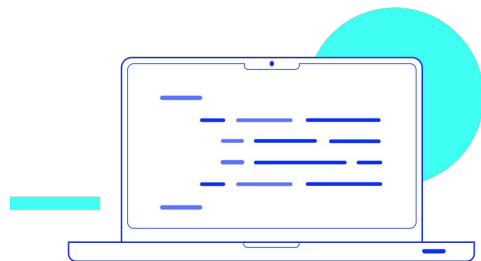
El modelo tiende a evolucionar en la fase de construcción a través de *feedbacks* que realizan los desarrolladores.



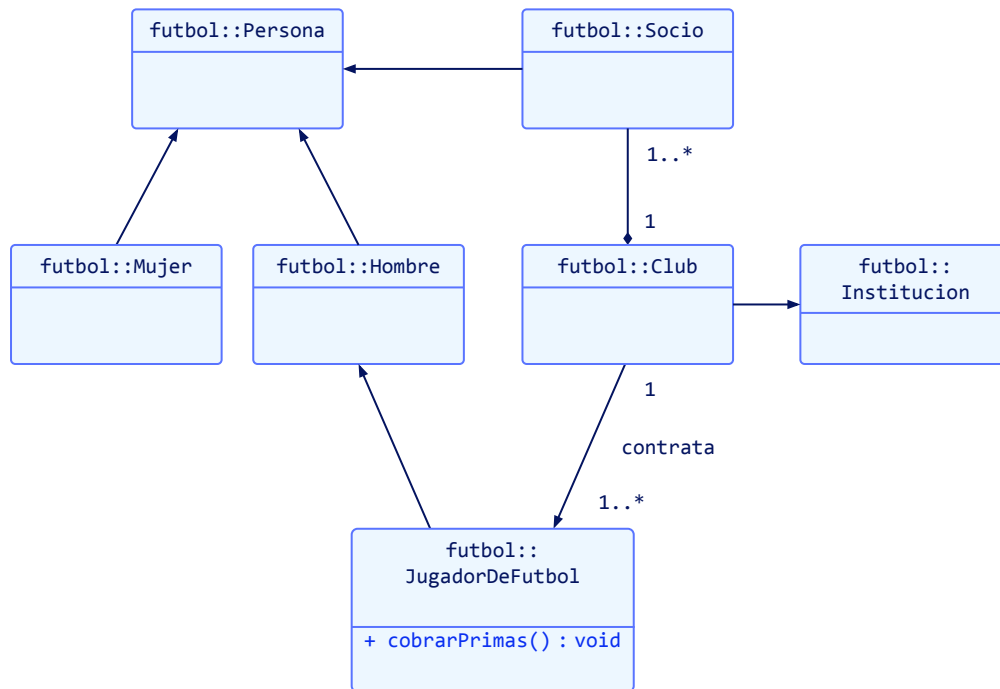
Diseño de Base de Datos

También es posible realizar el modelo de datos o diseño de una base de datos a través de un diagrama de clases. En este caso **las clases representan las tablas y los atributos representan los campos**.

Para esto es necesario utilizar estereotipos (ver Sección Conceptos Generales), determinando el estereotipo <<table>> para las tablas, el estereotipo <<column>> para los campos, y otros estereotipos posibles como <<PK>> para las claves primarias y <<FK>> para las claves foráneas, entre otros.

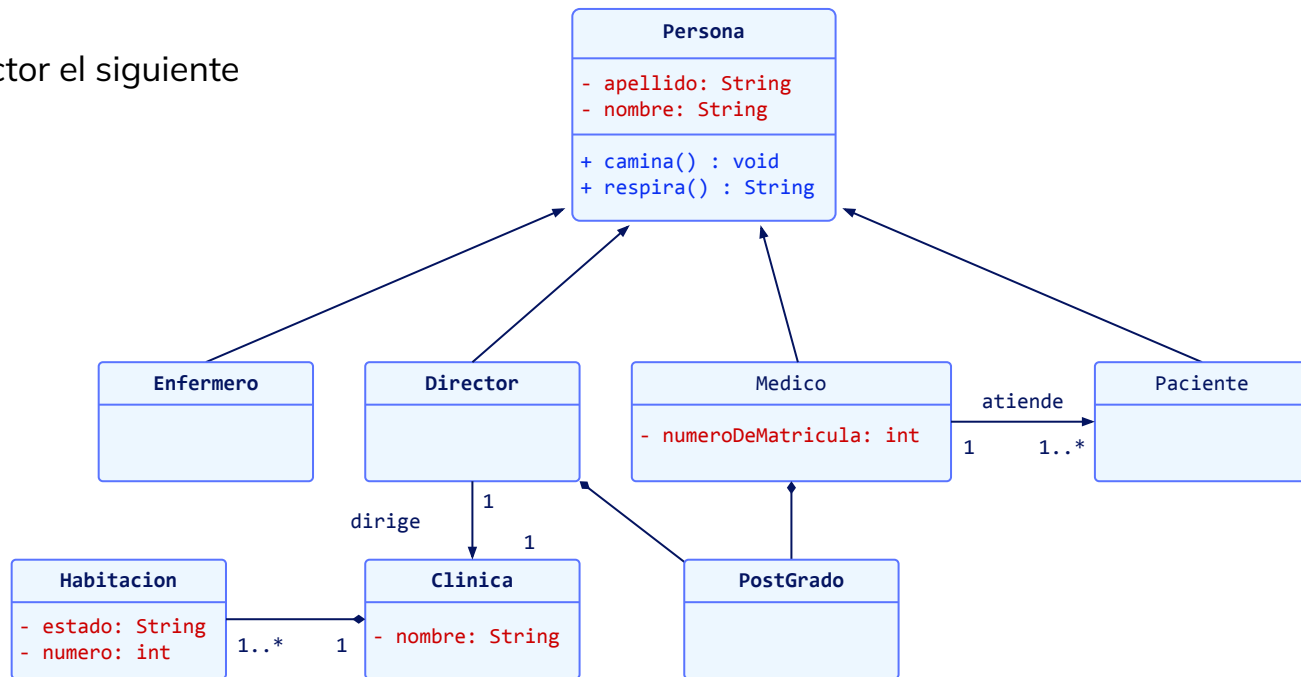


Ejemplo



Tutorial

Realice junto al instructor el siguiente
Diagrama de Clases:



**¡Sigamos
trabajando!**