Tecnicatura Universitaria en Programación
Programación II

UNIDAD N° 3 y 4: Programación Orientada a Objetos - Apunte 1

Índice

OOP

Introducción

Si queremos modelar en un estilo orientado a objetos, primero debemos aclarar qué significa Orientación a objetos. La introducción de la orientación a objetos se remonta a la década de 1960, cuando se presentó el lenguaje de simulación SIMULA, basado en un paradigma que era lo más natural posible para los humanos para describir el mundo.

No existe una única definición para la orientación a objetos. Sin embargo, existe un consenso general sobre las propiedades que caracterizan la orientación a objetos.

La programación orientada a objetos (POO) es uno de los enfoques más eficaces para escribir software. En la programación orientada a objetos, se escriben **clases** que representan cosas y situaciones del mundo real y se crean **objetos** basados en estas clases. Cuando escribes una clase, defines el comportamiento general que puede tener toda una categoría de objetos.

Cuando crea objetos individuales de la clase, cada objeto se equipa automáticamente con el comportamiento general; Luego puedes darle a cada objeto los rasgos únicos que desees. Crear un objeto a partir de una clase se llama creación de **instancias** y se trabaja con instancias de una clase.

También se pueden definir clases que amplíen la funcionalidad de clases existentes, de modo que clases similares puedan compartir una funcionalidad común y puedas hacer más con menos código.

Aprender sobre programación orientada a objetos le ayudará a ver el mundo como lo ve un programador. Le ayudará a comprender su código, no solo lo que sucede línea por línea, sino también los conceptos más amplios detrás de él. Conocer la lógica detrás de las clases lo capacitará para pensar de manera lógica, de modo que pueda escribir programas que aborden de manera efectiva casi cualquier problema que encuentre.

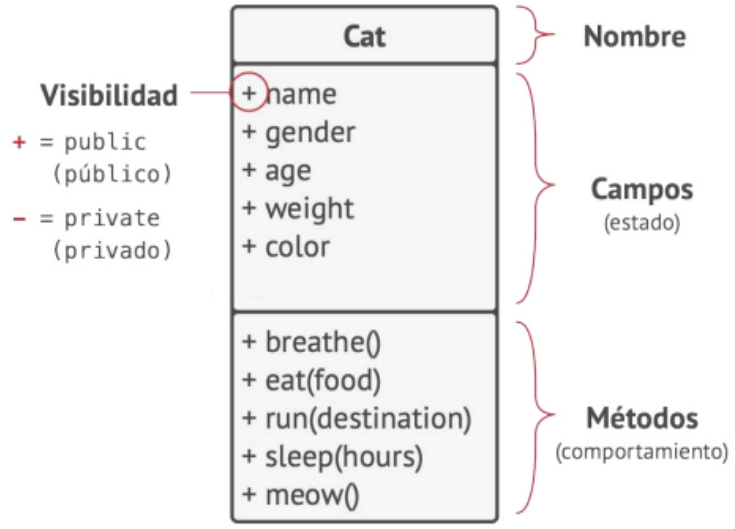
Conceptos básicos de POO

Las clases describen los atributos y el comportamiento de un conjunto de objetos de manera abstracta y así agrupa características comunes de los objetos del mundo real.

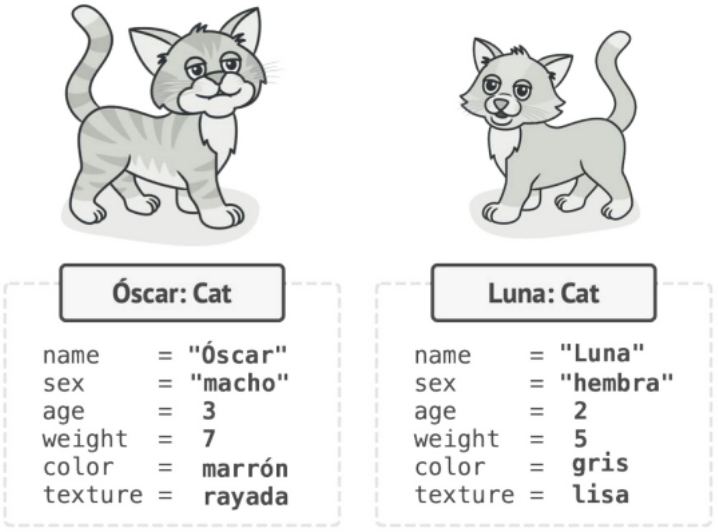
Digamos que tienes un gato llamado Felix. Felix es un **objeto**, una instancia de la **clase** Cat. Cada gato tiene varios **atributos** estándar: nombre, sexo, edad, peso, color, comida favorita, etc. Estos son los campos de la clase.
Además, todos los gatos se comportan de forma similar: respiran, comen, corren, duermen y maúllan. Estos son los **métodos** de la clase.

La información almacenada dentro de los campos del objeto suele denominarse **estado**. El estado de un objeto viene determinado por los valores que toman sus datos o atributos.

Colectivamente, podemos referirnos a los campos y los métodos como los **miembros** de una clase.



Esto es un diagrama de clases en UML. Más adelante se introducirá el tema.



Ejemplos de objetos de la clase Cat.

Por lo tanto, una clase es como una plantilla que define la estructura de los objetos, que son instancias concretas de esa clase.

La identidad permite diferenciar los objetos de modo no ambiguo independientemente de su estado. Es posible distinguir dos objetos en los cuáles todos sus atributos sean iguales. Cada objeto posee su propia identidad de manera implícita. Cada objeto ocupa su propia posición en la memoria de la computadora.

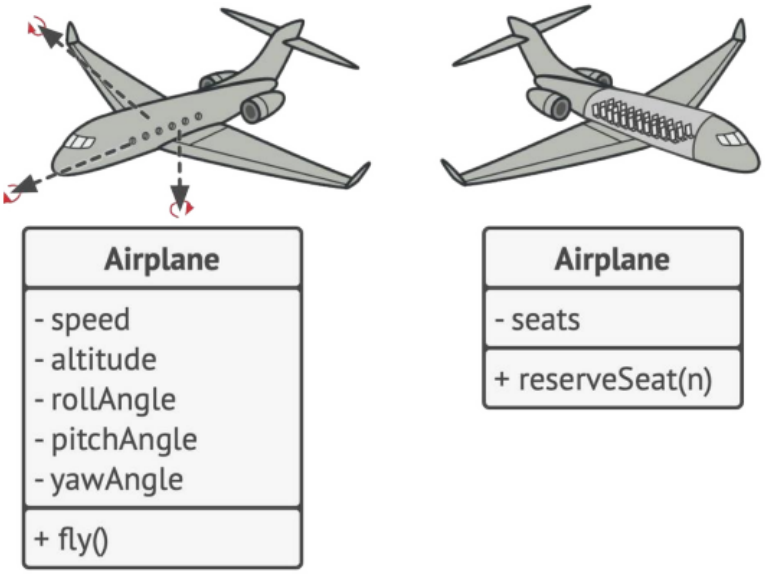
Propiedades fundamentales de la programación orientada a Objetos

Abstracción

La mayoría de las veces, cuando creas un programa con POO, das forma a los objetos del programa con base a objetos del mundo real. Sin embargo, los objetos del programa no representan a los originales con una precisión del 100 % (y rara vez es necesario que lo hagan).

En su lugar, tus objetos tan solo copian atributos y comportamientos de objetos reales en un contexto específico, ignorando el resto.

Por ejemplo, una clase Avión probablemente podría existir en un simulador de vuelo y en una aplicación de reserva de vuelos. Pero, en el primer caso, contendría información relacionada con el propio vuelo, mientras que en la segunda clase sólo habría que preocuparse del mapa de asientos y de los asientos que estén disponibles.



Distintos modelos del mismo objeto del mundo real.

La Abstracción es el modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.

Encapsulación

Para arrancar el motor de un auto, tan solo debes girar una llave o pulsar un botón. No necesitas conectar cables bajo el capó, rotar el cigüeñal y los cilindros, e iniciar el ciclo de potencia del motor. Estos detalles se esconden bajo el capó del auto. Sólo tienes una **interfaz** simple: un interruptor de encendido, un volante y unos pedales. Esto ilustra el modo en que cada objeto cuenta con una interfaz: una parte pública de un objeto, abierta a interacciones con otros objetos.

La encapsulación es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz limitada al resto del programa.

Encapsular algo significa hacerlo privado y, por ello, accesible únicamente desde dentro de los métodos de su propia clase.

La encapsulación oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior, por lo que se denomina también ocultación de datos.

Herencia

La herencia es la capacidad de crear nuevas clases sobre otras existentes. La principal ventaja de la herencia es la reutilización de código. Si quieres crear una clase ligeramente diferente a una ya existente, no hay necesidad de duplicar el código.

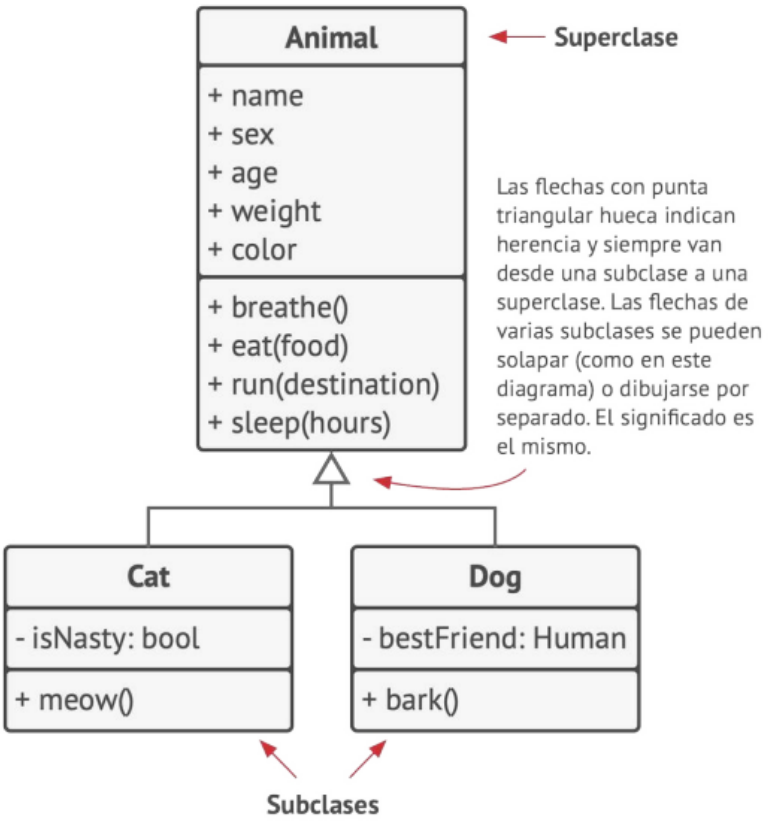
En su lugar, extiendes la clase existente y colocas la funcionalidad adicional dentro de una subclase resultante que hereda los campos y métodos de la superclase.

La consecuencia del uso de la herencia es que las subclases tienen la misma interfaz que su clase padre. No puedes esconder un método en una subclase si se declaró en la superclase.

En la mayoría de los lenguajes de programación una subclase puede extender una única superclase.

Por ejemplo, digamos que tu vecino tiene un perro llamado Fido. Resulta que perros y gatos tienen mucho en común: nombre, sexo, edad y color, son atributos tanto de perros como de gatos. Los perros pueden respirar, dormir y correr igual que los gatos, por lo que podemos definir la clase base Animal que enumerará los atributos y comportamientos comunes.

Una clase padre, como la que acabamos de definir, se denomina superclase. Sus hijas son las subclases. Las subclases heredan el estado y el comportamiento de su padre y se limitan a definir atributos o comportamientos que son diferentes. Por lo tanto, la clase Cat contendrá el método maullar y, la clase Dog, el método ladrar .



Asumiendo que tenemos una tarea relacionada, podemos ir más lejos y extraer una clase más genérica para todos los Organismo vivos, que se convertirá en una superclase para Animal y Planta . Tal pirámide de clases es una jerarquía. En esta jerarquía, la clase Cat lo hereda todo de las clases Animal y Organismo.

Las subclases pueden sobrescribir el comportamiento de los métodos que heredan de clases padre. Una subclase puede sustituir completamente el comportamiento por defecto o limitarse a mejorarlo con líneas de código extras.

Polimorfismo

El polimorfismo es la propiedad que permite tener el mismo nombre de método en clases diferentes y que actúe de modo diferente en cada una de ellas.

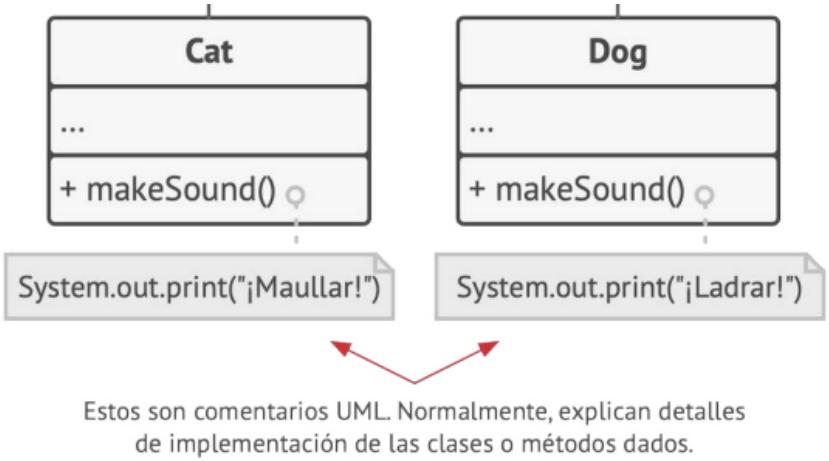
Veamos algunos ejemplos con animales. La mayoría de los animales pueden emitir sonidos. Podemos anticipar que todas las clases necesitarán un método emitirSonido.

Imagina que ponemos varios gatos y perros dentro de una gran bolsa. Después, con los ojos tapados, vamos sacando los animales de la bolsa, de uno en uno. Al sacar un animal, no sabemos con seguridad lo que es. Pero el animal emitirá un sonido específico, dependiendo de su clase concreta.

El programa no conoce el tipo concreto del objeto contenido dentro de la variable a , pero, gracias al mecanismo especial llamado polimorfismo, el programa puede rastrear la clase

del objeto cuyo método está siendo ejecutado, y ejecutar el comportamiento adecuado.

El polimorfismo es la capacidad que tiene un programa de detectar la verdadera clase de un objeto e invocar su implementación.



Modelado de Aplicaciones: UML - Diagrama de clase

El Lenguaje Unificado de Modelado (UML, Unified Model Language), es el lenguaje estándar de modelado para desarrollo de sistemas y de software. UML se ha convertido de facto en el estándar para modelado de aplicaciones software y ha crecido su popularidad en el modelado de otros dominios. Tiene una gran aplicación en la representación y modelado de la información que se utiliza en las fases de análisis y diseño. En diseño de sistemas, se modela por una importante razón: gestionar la complejidad. Un modelo es una abstracción de cosas reales.

Cuando se modela un sistema, se realiza una abstracción ignorando los detalles que sean irrelevantes. El modelo es una simplificación del sistema real. Con un lenguaje firmal de modelado, el lenguaje es abstracto aunque tan preciso como un lenguaje de programación. Esta precisión permite que un lenguaje sea legible por la máquina, de modo que pueda ser interpretado, ejecutado y transformado entre sistemas.

Para modelar un sistema de modo eficiente, se necesita una cosa muy importante: un lenguaje que pueda describir el modelo. ¿Qué es UML? UML es un lenguaje. Esto significa que tiene tanto sintaxis como semántica y se compone de: pseudocódigo, código real, dibujos, programas, descripciones... Los elementos que constituyen un lenguaje de modelado se denominan notación.

El bloque básico de construcción de UML es un diagrama. Existen tipos diferentes,

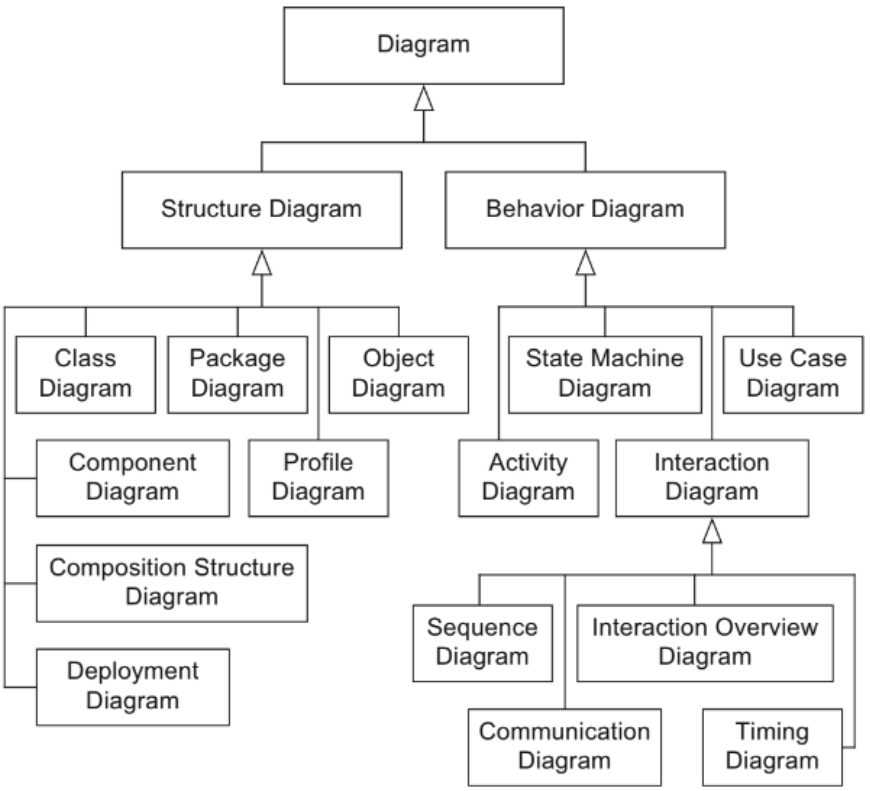


Figure 2.1
UML diagrams

Diagramas de estructura

UML ofrece siete tipos de diagramas para modelar la estructura de un sistema desde diferentes perspectivas. En estos diagramas no se considera el comportamiento dinámico de los elementos en cuestión (es decir, sus cambios en el tiempo).

Diagrama de clases

Usamos el diagrama de clases para modelar la estructura estática de un sistema, por lo que el diagrama de clases describe los elementos del sistema y las relaciones entre ellos. Estos elementos y las relaciones entre ellos no cambian con el tiempo.

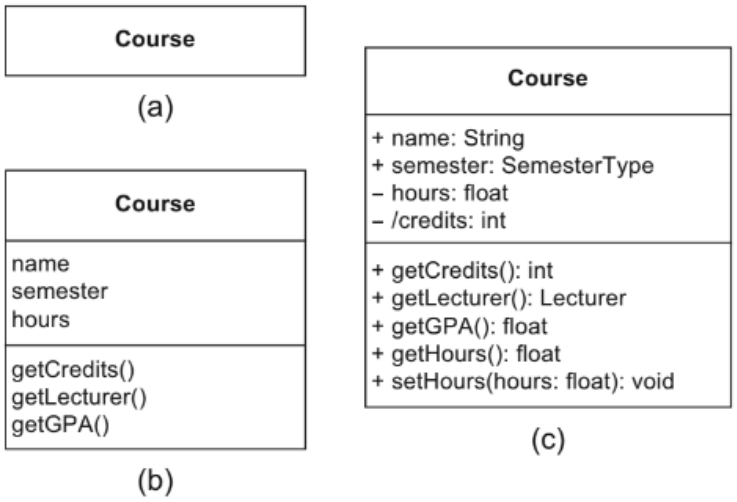
Por ejemplo, los estudiantes tienen un nombre y un número de legajo y cursan varias materias. Esta sentencia cubre una pequeña parte de la estructura universitaria y no pierde validez ni siquiera con el paso de los años. Son sólo los estudiantes los que cambian.

El diagrama de clases es sin duda el diagrama UML más utilizado. Se aplica en varias fases del proceso de desarrollo de software. El nivel de detalle o abstracción del diagrama de clases es diferente en cada fase. En las primeras fases del proyecto, un diagrama de clases le permite crear una vista conceptual del sistema y definir el vocabulario que se utilizará. Luego puede refinar este vocabulario en un lenguaje de programación hasta el punto de implementación. En el contexto de la programación orientada a objetos, el diagrama de clases visualiza las clases que componen un sistema de software y las relaciones entre estas clases. Debido a su simplicidad y popularidad, el diagrama de clases es ideal para bocetos rápidos. Sin embargo, también puedes usarlo para generar código de programa automáticamente. En la práctica, el diagrama de clases también se utiliza a menudo con fines de documentación.

Nomenclatura del diagrama de clases

En un diagrama de clases, una clase está representada por un rectángulo que se puede subdividir en varios compartimentos. El primer compartimento debe contener el nombre de la clase, que generalmente comienza con una letra mayúscula y se coloca centrado en negrita. Según las convenciones de nomenclatura comunes, los nombres de clases son sustantivos singulares.

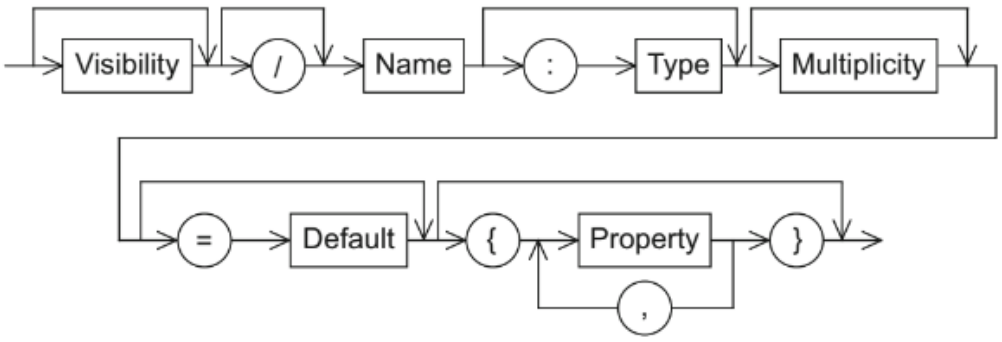
El segundo compartimento del rectángulo contiene los atributos de la clase, y el tercer compartimento contiene los métodos.



Si no se incluye información específica en el diagrama, esto no significa que no exista; simplemente significa que esta información no es relevante en este momento o no se incluye por razones prácticas, por ejemplo, para evitar que el diagrama se vuelva demasiado complicado.

Atributos

Un atributo tiene al menos un nombre. Y su sintaxis de definición es:



La especificación de una barra diagonal / antes del nombre de un atributo indica que el valor de este atributo se deriva de otros atributos. Un ejemplo de atributo derivado es la edad de una

persona, que se puede calcular a partir de la fecha de nacimiento.

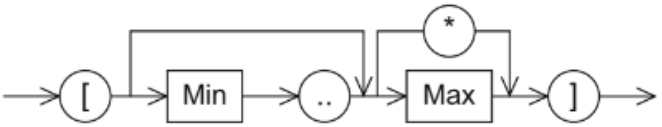
El tipo de atributo se puede especificar después del nombre usando **: Tipo**. Los posibles tipos de atributos incluyen tipos de datos primitivos del lenguaje.

Para definir un valor predeterminado para un atributo, especifique **= default** , donde el valor predeterminado es un valor o expresión definido por el usuario. El sistema utiliza el valor predeterminado si el valor del atributo no lo establece explícitamente el usuario.

Puede especificar propiedades adicionales del atributo entre corchetes. Por ejemplo, la propiedad **{readOnly}** significa que el valor del atributo no se puede cambiar una vez que se ha inicializado.

Multiplicidad

La multiplicidad de un atributo indica cuántos valores puede contener un atributo. Esto Generalmente se codifica como una tipo iterable dependiendo los tipos del lenguaje de programación.



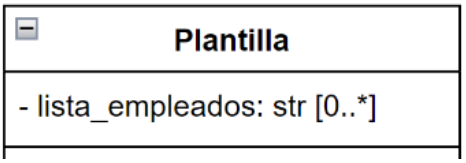
La multiplicidad se muestra como un intervalo encerrado entre corchetes en la forma [mínimo... máximo], donde mínimo y máximo son números naturales que indican los límites superior e inferior del intervalo. El valor del mínimo debe ser menor o igual que el valor del máximo. Si no hay un límite superior para el intervalo, éste se expresa con un asterisco ∗.

Si el mínimo y el máximo son idénticos, no es necesario especificar el mínimo ni los dos puntos; Por ejemplo [5].

La expresión [∗] es equivalente a [0..∗]. Si no especifica una multiplicidad para un atributo, se supone que el valor 1 es el predeterminado, lo que especifica un atributo de un solo valor.

Si un atributo puede adoptar múltiples valores, se podrá codificar por ejemplo en Python cómo:
Un conjunto (sin orden fijo de elementos, sin duplicados)
Una lista (orden fijo, posibles duplicados)

Puede realizar esta especificación combinando propiedades {non-unique} y {unique}, que definen si se permiten o no duplicados, y {ordered} y {unordered}, que fuerzan un orden fijo de los valores de los atributos.



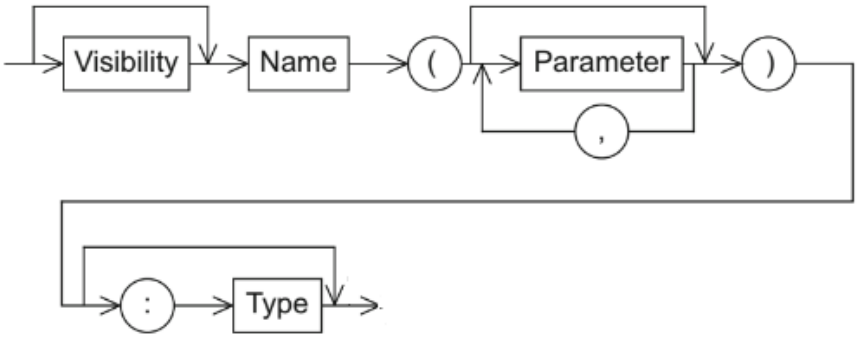
Por ejemplo, tenemos a la clase Plantilla que contiene a un atributo privado lista_empelado que es una colección de strings de 0 a un número indefinido.

Comportamiento

Las operaciones se caracterizan por su nombre, sus parámetros y el tipo de su valor de retorno. Cuando se llama a una operación en un programa, se ejecuta el comportamiento asignado a esta operación. En los lenguajes de programación, una operación corresponde a una declaración de método.

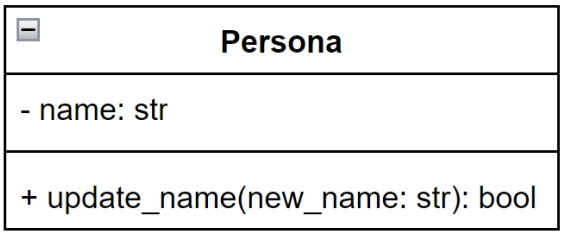
El diagrama de clases no es adecuado para describir el comportamiento de objetos en detalle ya que sólo modela firmas de las operaciones que proporcionan los objetos; no modela cómo se implementan realmente estas operaciones. UML ofrece diagramas de comportamiento especiales para representar la implementación de operaciones, por ejemplo el diagrama de actividades.

La sintaxis para definir un método en el diagrama de clase es:



En un diagrama de clases, el nombre del método va seguido de una lista de parámetros entre paréntesis. La lista en sí puede estar vacía. Un parámetro se representa de manera similar a un atributo. La única información obligatoria es el nombre del parámetro. La adición de un tipo y un valor predeterminado son opcionales.

El valor de retorno es opcional y se especifica con el tipo de valor de retorno.



Por ejemplo, el comportamiento definido para la clase Persona, tenemos el método update_name, que recibe cómo único parámetro, new_name de tipo string y especifica el nuevo nombre de una persona. El valor de retorno tiene el tipo booleano. Si se devuelve verdadero, el cambio de nombre se realizó correctamente; de lo contrario, se devuelve falso.

Visibilidad

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), deben colocarse la anotación antes del nombre de los miembros.

public	**+**	Todos los miembros declarados cómo públicos son de libre acceso desde cualquier otra parte de un programa.
private	**-**	Todos los miembros declarados cómo privados no son accesibles por fuera de la clase.
protected	**#**	Todos los miembros declarados cómo protegidos son accesibles por la clase en que fueran declarados y todas las subclases.

Variables de clase y métodos de clase

Los atributos normalmente se definen a nivel de instancia. Si, por ejemplo, una clase se realiza en un lenguaje de programación, se reserva memoria para cada atributo de un objeto cuando se crea. Estos atributos también se denominan variables de instancia o atributos de instancia.

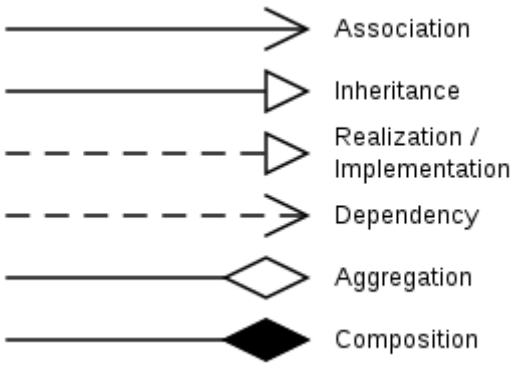
A diferencia de las variables de instancia, las variables de clase se crean sólo una vez para una clase y no por separado para cada instancia de esta clase. Estas variables también se denominan atributos estáticos o atributos de clase.

En el diagrama de clases, los atributos estáticos están subrayados, al igual que los métodos estáticos.

Los métodos estáticos, también llamados métodos de clase, se pueden utilizar si no se creó ninguna instancia de la clase correspondiente. Ejemplos de operaciones estáticas son funciones matemáticas como sin(x) o constructores. Los constructores son funciones especiales llamadas para crear una nueva instancia de una clase.

Relaciones

Las asociaciones entre clases modelan posibles relaciones, conocidas como vínculos, entre instancias de las clases. Describen qué clases son posibles socios de comunicación. Si sus atributos y operaciones tienen las visibilidades correspondientes, los socios de comunicación pueden acceder a los atributos y operaciones de cada uno. En un diagrama de clases puede verse las asociaciones entre las clases.



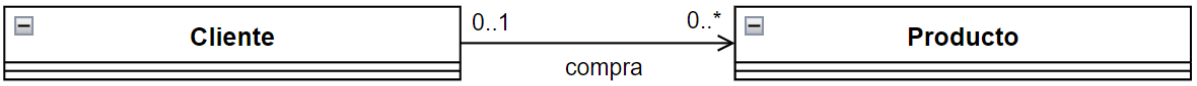
0	Sin casos (raro)
0..1	Ninguna instancia o una instancia
1	exactamente una instancia
1..1	exactamente una instancia
0..*	Cero o más instancias
*	Cero o más instancias
1..*	Una o más instancias

Multiplicidad en las asociaciones entre clases

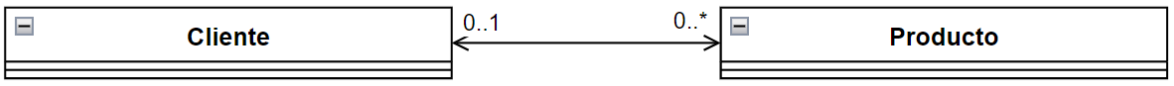
Asociaciones

Una asociación binaria nos permite asociar las instancias de dos clases entre sí. Las relaciones se muestran como bordes (línea continua) entre las clases de socios involucradas. El borde se puede etiquetar con el nombre de la asociación seguido opcionalmente de la dirección de lectura, un pequeño triángulo negro. La dirección de lectura está dirigida hacia un extremo de la asociación y simplemente indica en qué dirección el lector del diagrama debe "leer" el nombre de la asociación. Además generalmente se coloca la multiplicidad.

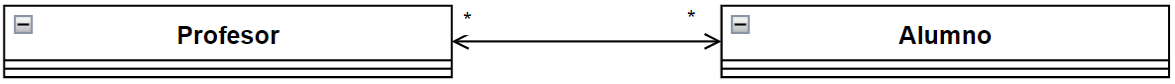
En orientación a objetos, el comportamiento de un sistema se define en términos de interacciones entre objetos, es decir, intercambios de mensajes. "Enviar un mensaje" habitualmente resulta en la invocación de una operación en el receptor. Las asociaciones son necesarias para la comunicación, ya que los mensajes son enviados a través de las asociaciones; sin asociaciones, los objetos quedarían aislados, incapaces de interactuar.
En los lenguajes de programación orientados a objetos podemos codificar estas relaciones cómo un atributo de tipo objeto de la relación.



En este caso el cliente conoce los productos que compró, por ejemplo en un resumen del pedido. Pero el Producto no conoce el cliente que lo compró.



En este caso el producto conoce al cliente que lo compró y por lo tanto, de una lista de productos vendidos es posible obtener una lista de los clientes que compran más de x cantidad de productos mensuales.



En este caso un profesor accede a todos sus alumnos, y un alumno puede acceder a todos sus profesores.

Dependencia

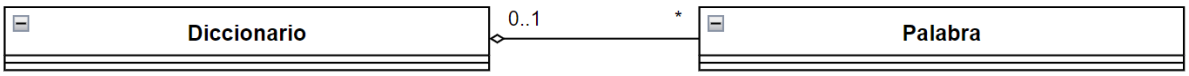
Es una forma de asociación que especifica algún tipo de dependencia entre dos clases, donde un cambio en la clase de la cual se depende puede afectar a la clase dependiente, pero no necesariamente a la inversa. En UML se representa como una asociación pero, en lugar de usar una línea sólida se utiliza una línea punteada. Puede agregarse una flecha para indicar dependencia asimétrica.
En la mayoría de los casos, las dependencias se reflejan en los métodos de una clase que utilizan el objeto de otra clase como parámetro .
Una relación de dependencia es una relación de "uso". Un cambio en una cosa en particular puede afectar a otras cosas que la usan, y usar una dependencia cuando es necesario indicar que una cosa usa otra.

La clase CronogramaMateria tiene una dependencia con el CalendarioAcademico ya que es pasado cómo parámetro a la llamada del método realizar_cronograma. Un cambio en la clase CalendarioAcademico podría impactar en la implementación del método.

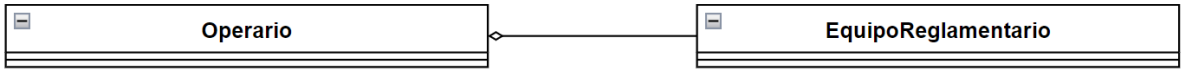
La clase Sueldo depende de la clase Calculadora, ya que para realizar comportamiento cómo por ejemplo calcular el sueldo a abonar, llama al método calcular_ret_ganancias.

Agregaciones

La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil).
Las relaciones agregadas también representan la relación entre el todo y una parte de la clase, los objetos miembros son parte del objeto general, pero el objeto miembro puede existir independientemente del objeto general.
La destrucción del compuesto no conlleva la destrucción de los componentes. Habitualmente se da con mayor frecuencia que la composición.
La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el "todo".



Por ejemplo, las palabras son parte de un diccionario, pero existen por fuera de él.



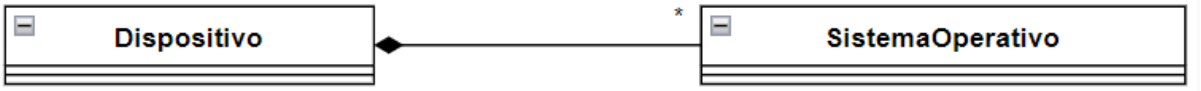
Por ejemplo, un operario y su equipo reglamentario son parte de la relación todo/parte, pero se pueden separar.

Estas relaciones comúnmente se codifican cómo un parámetro pasado en el constructor de una clase. A veces se codifica al igual que las asociaciones.

Composiciones

Composición es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora. Los componentes constituyen una parte del objeto compuesto. La supresión del objeto compuesto conlleva la supresión de los componentes.
El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el "todo".

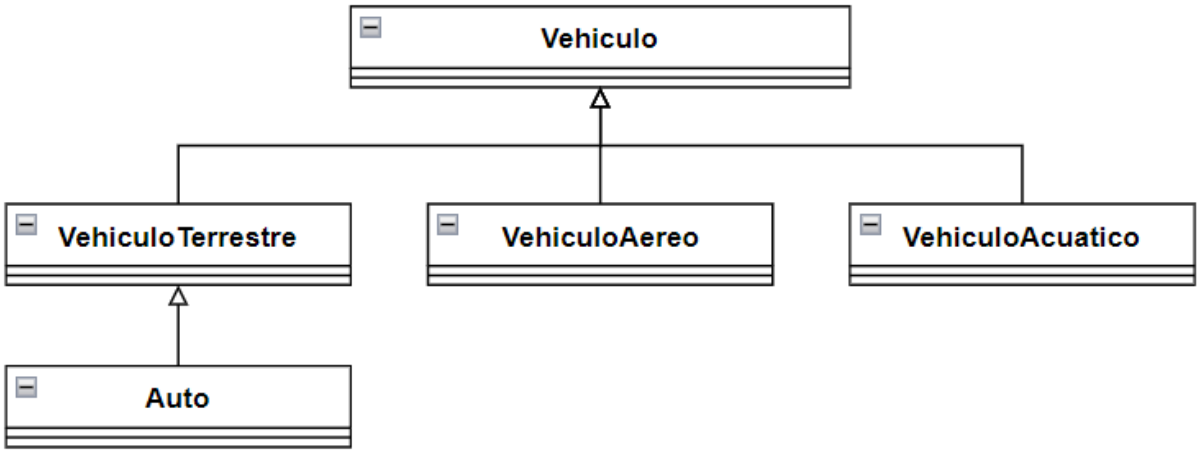
Estas relaciones comúnmente se codifican instanciando al objeto parte dentro de la clase todo, ya sea en el constructor o en algún método.



Imaginemos que una clase Dispositivo está compuesto de un sistema operativo dedicado, que fue diseñado específicamente para el dispositivo. Podemos decir que un celular está compuesto de su SistemaOperativo y que si el Dispositivo desaparece, también desaparecerá el SistemaOperativoDedicado. Entonces, aquí hay una relación de composición.

Herencia

La relación de generalización expresa que las características (atributos y operaciones) y asociaciones que se especifican para una clase general (superclase) se pasan a sus subclases. Por tanto, la relación de generalización también se denomina herencia. Esto significa que cada instancia de una subclase es simultáneamente una instancia indirecta de la superclase. La subclase "posee" todos los atributos de instancia y atributos de clase y todas las operaciones de instancia y operaciones de clase de la superclase siempre que no hayan sido marcadas con visibilidad privada.
La subclase también puede tener otros atributos y operaciones o entrar en otras relaciones independientemente de su superclase. En consecuencia, las operaciones que se originan en la subclase o superclase se pueden ejecutar directamente en la instancia de una subclase.
Una relación de generalización está representada por una flecha con punta triangular desde la subclase a la superclase.

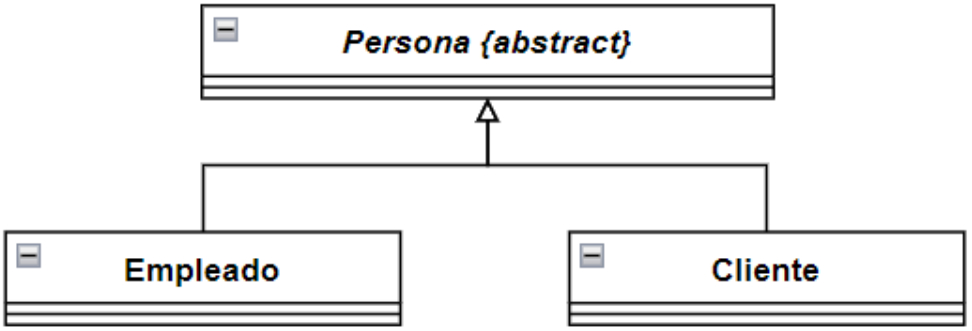


Esto se codifica extendiendo desde la subclase a la clase de la cual se quiere heredar. De esta forma generamos una jerarquía de clases.

Estas son todas las relaciones que vamos a utilizar.

Clases abstractas

Las clases de las que no se pueden crear instancias por sí mismas se modelan como clases abstractas. Estas son clases para las cuales no hay objetos; sólo se pueden crear instancias de sus subclases. Las clases abstractas se utilizan exclusivamente para resaltar características comunes de sus subclases y, por lo tanto, son sólo útiles en el contexto de relaciones de generalización. Las operaciones de clases abstractas también pueden etiquetarse como abstractas. Una operación abstracta La operación abstracta no ofrece ninguna implementación en sí misma. Sin embargo, requiere una implementación en las subclases concretas. Las operaciones que no son abstractas transmiten su comportamiento a todas las subclases.
Las clases abstractas y las operaciones abstractas están escritas en cursiva o indicadas mediante la especificación de la palabra clave {abstract} antes de su nombre. En particular, en los diagramas de clases producidos manualmente, se recomienda el uso de la segunda notación alternativa, ya que la escritura en cursiva es difícil de reconocer.



Supongamos que estamos diseñando un sistema para una empresa, entonces las clases Empleado y Cliente heredan de la clase abstracta Persona. No tiene sentido para el contexto del problema crear un objeto de la clase Persona, ya que se acotó el contexto a las personas que o son clientes o son empleados.

Bibliografía

https://docs.python.org/3/tutorial/classes.html
Seidl, M., Huemer, M. S. C., & Kappel, G. (2012). UML@ Classroom An Introduction to Object-Oriented Modeling.
Dive Into Design Patterns
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming. no starch press.

Versiones

Fecha	Versión
30/08/2023	1.0

Autores

María Mercedes Valoni