Las gotas de UML que hacen falta[1]

por Lucas Videla

Introducción

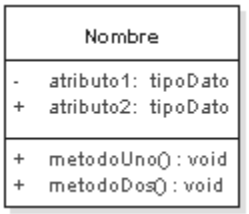
Existe una materia en la carrera de Ingeniería en Informática que se centra sobre el UML y, dada su existencia, este apunte no pretende ser reemplazo de esos conocimientos, sino más bien introducción para quienes no la han cursado aún.
Realmente el contenido será conciso y concreto sobre los puntos importantes al momento de encarar los trabajos del taller. Es necesario que estemos de acuerdo sobre ciertas convenciones, para que compartamos un lenguaje común que permita un mutuo entendimiento.
A partir del momento en que se libera este artículo, se da por sentado que la forma de modelar el software que se construya estará documentado mediante estas instrucciones. Cualquier duda será respondida.

¿Qué es UML?

UML es la sigla de **Unified Modeling Language**, es decir, el Lenguaje de Modelado Unificado. Es un lenguaje estándar para escribir planos de software. Es rápido de ver, interpretar y visualizar dado un problema particular, y sirve, como su nombre lo indica, para unificar el modo de modelar software.
Se vio la necesidad de la creación de este lenguaje dada la ambigüedad y falta de coordinación entre las distintas partes de los equipos de desarrollo.
UML es un lenguaje cuyos símbolos son gráficos (e independiente de cualquier lenguaje de programación), con carteles explicativos añadidos sólo cuando es necesario. Posee muchos elementos, pero los que aquí usaremos son los que atañen a los llamados *Diagramas de Clases*.

Diagrama de Clases

El diagrama de clases, como su nombre lo indica, sirve para dar un panorama general sobre las clases existentes en un proyecto software y, fundamentalmente, la relación que hay entre ellas.
Permite visualizar en forma estática la vinculación existente, además de los atributos, métodos y responsabilidades propias a cada clase.
El elemento principal es la Clase, que es un rectángulo dividido en tres partes. Veamos:

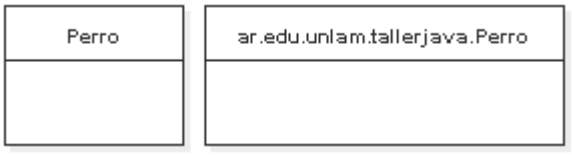


Cada una de esas partes se explicará a continuación.

Nombre

Cada clase debe tener un nombre que la distinga. Es una cadena de texto, que puede o no incluir la ruta del paquete al que pertenece. Un ejemplo podría ser la clase ArrayList, o con su paquete correspondiente java.util.ArrayList.
Se utilizan generalmente nombres cortos y muy representativos, comenzando con una letra mayúscula, y solamente mayúsculas adicionales en caso de que sean varias palabras. Ejemplos: *Perro, OrdenamientoBurbuja, CuentaBancaria*, pero nunca ~~*perro, ORDENAMIENTOBURBUJA*~~ o ~~*Cuentabancaria*~~.

En el diagrama, se adorna de diferentes maneras, según sea el nombre de una clase, una clase abstracta o una interfaz, como veremos más adelante.



Dos versiones sobre la misma clase, abreviado o con nombre de ruta

Atributos

Son las propiedades de las clases. Describe la variabilidad sobre el patrón base que establece la clase. Por ejemplo, un atributo del ser humano es el color de cabello. Si habláramos de la clase SerHumano, el atributo colorPelo podría tomar valores como morocho, rubio, etcétera. De este modo, se varía sobre la misma clase (por tener diferentes colores de cabello no dejan de ser seres humanos).

Esta propiedad debe ser compartida por todos los objetos de la clase. En algún momento, el objeto de la clase tendrá valores asignados a esos atributos, que le servirán para algún fin específico.

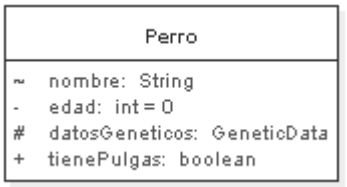
Los atributos son cadenas de texto (independientemente del tipo de dato que representen), con nombres representativos. Se pone en mayúscula cada primera letra de las palabras que nombran al atributo, salvo la primera (para distinguirlos de las clases). Por ejemplo, *colorPelo*, *edad*, *numeroObraSocial*, pero nunca ~~*COLORPELO*~~, ~~*Edad*~~ o ~~*numeroobrasocial*~~.

Usualmente se acompaña por izquierda con un símbolo que identifica la visibilidad del atributo (concepto que no se explicará aquí), pudiendo tomarse por estándar el signo – (menos) para los atributos privados, + (más) para los atributos públicos, y # (numeral) para los atributos protegidos. En Java, particularmente, si se deja el atributo acompañado por un ~ (ñuflo), se considera de orden "paquete", es decir, tiene una visibilidad análoga a las variables y funciones amigas de C.

Por derecha se acompaña al nombre del atributo con un par de dos puntos y el tipo de dato que representa ese atributo, sea flotante, entero, carácter, cadena...

En caso de estar asignado un valor por defecto a la variable, se especifica también en ese lugar, luego de un signo = (igual).

Veamos un ejemplo de una clase con atributos de los más variados:



Vamos a explicar un poco el diagrama. Debe considerarse que las visibilidades se han puesto de una forma quizás lógica, quizás no, simplemente para mostrar la variedad que existe.

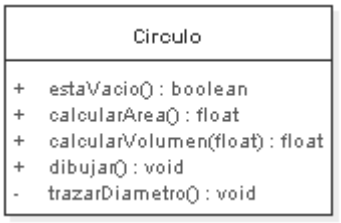
Tenemos una clase Perro, la cual tiene atributos. El nombre, que es de tipo String, cuya visibilidad es de paquete (todo quien sabe que está ahí conoce su nombre). La edad, un entero privado, que sólo conoce el perro y sólo él puede modificarla, inicializada en cero (cuando nace).

Los datos genéticos son de visibilidad protegida, por lo que se heredan como privados (el modelado de clases no debería ser de ese modo, pero el ejemplo vale, simplemente para mostrar "intuitivamente" la visibilidad).

La cantidad de pulgas es un dato público, booleano (todos nos damos cuenta cuando un perro tiene pulgas).

Cabe destacar nuevamente que esta modelación de la clase perro es caprichosa, sólo a los efectos de explicar la parte de atributos y no necesariamente deba ser correcta.

Métodos
Son las implementaciones de los servicios que una clase debe brindar, para cumplir su función: su comportamiento. Es una abstracción de algo que la clase puede hacer. Las operaciones se sirven de o cambian los estados del objeto particular **sobre el que se efectúa la operación** (encapsulamiento)**.**
Se nombran del mismo modo que los atributos, además de utilizar a izquierda los caracteres que identifican la visibilidad del método.
Por derecha, generalmente se acompañan de un paréntesis que puede o no tener dentro los datos y tipos de datos que recibirán, separados por dos puntos respectivamente y por comas entre sí.
Luego de cerrado el paréntesis pueden llevar un signo de dos puntos con el tipo de dato que el método devolverá.
Como ejemplo, podemos proponer el siguiente:



En este caso estamos ante una clase Círculo, que tiene una serie de métodos públicos sin parámetros como estaVacio, calcularArea y dibujar, devolviendo datos sólo los dos primeros.
El método público calcularVolumen, necesita de un flotante para calcular el correspondiente volumen, y es la altura del cilindro asociado.
El método privado trazarDiametro puede ser de utilidad a algún otro método público para desarrollar su tarea, pero poca utilidad puede tener para quien acceda al objeto desde fuera de la clase.

Responsabilidades
La responsabilidad es el contrato (el conjunto de obligaciones) que una clase debe cumplir. Para ello se sirve de los métodos y atributos.
Generalmente no se modelan, debido a que su existencia se denota por el nombre que usualmente se le asigna a los métodos. Pero, de ser necesario, se agrega un nuevo compartimiento al bloque de la clase, en el cual se coloca un listado de responsabilidades de la misma.
Se redactan en lenguaje informal, para el entendimiento de todos. No se ejemplificará ya que está implícito.

Abreviando
Aquellos atributos, métodos o responsabilidades que resulten obvias y/o triviales no deberían ser documentadas, sobre todo en caso de que el diagrama crezca considerablemente. Preferentemente se deja una lista con puntos suspensivos, simplemente consignando los más importantes.
Por ejemplo, el listado de *getters y setters*[1] no debe incluirse, ya que es reiterativo y trivial.

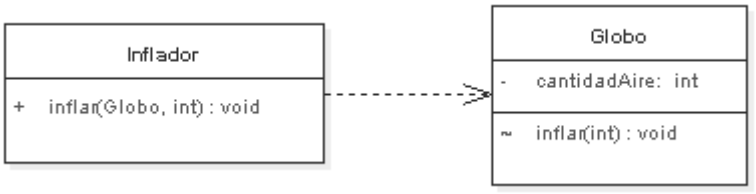
[1] Los **getters** y los **setters**, se recuerda, son los métodos públicos de toda clase, por los cuales se permite acceder de un **modo controlado** a los atributos (privados) de la misma, sin violar el principio de encapsulamiento. Los getters servirán para obtener el valor, y los setters para asignar uno nuevo.

¿Cómo modelar clases?
Intentar dar una respuesta a esta pregunta es quizás un poco ambicioso de nuestra parte, pero se pueden dar un par de consejos para que a la hora de elegir qué modelar, se lo haga de un modo aún no correcto, pero sí más adecuado. La práctica es la mejor forma de modelar correctamente.

Por empezar, se deben **identificar las abstracciones**. No es tarea fácil, y esto incluye identificar clases y el conjunto de responsabilidades (fundamentalmente) y métodos que servirán a esas responsabilidades, junto con los atributos correspondientes.
Muchas veces, un conjunto de atributos define una clase, o un grupo de responsabilidades es excusa para crear una clase que los agrupe (coherentemente). Por ejemplo, si tenemos que poder dar de altas alumnos en un registro de la biblioteca, así como libros... ¿cuáles serían las abstracciones necesarias para esas responsabilidades? Obviamente, empezaremos por el Alumno y el Libro, pero ambos pertenecen a una Colección o Listado, de Alumno y Libros respectivamente... ¿y la responsabilidad de agregarlos? ¿de quién es? Veamos el apartado siguiente.

Luego, se debe **asignar el grupo de responsabilidades correspondiente a cada clase**. *Es imperativo que las clases tengan las responsabilidades que le son propias, y no abarquen tareas de otras clases*.
Por ejemplo, teniendo las clases Globo e Inflador, de quién es la responsabilidad de inflar... ¿del globo o del inflador? Muy fácil es que nuestra mente piense rápidamente en un método llamado globo.inflar() ... pero ¿es esto correcto? Si necesitara controlar la cantidad y presión del aire que entra... ¿creen que sería responsabilidad del globo?

Se debe, por último, **seleccionar el grupo de atributos y métodos necesarios a cada clase para cumplir con las responsabilidades asignadas**. Es la parte más complicada, pero también la más intuitiva luego de un poco de práctica. En el ejemplo anterior, la cantidad de aire que tiene el globo, sería un atributo del globo. Pero para acceder a él (encapsulamiento), se necesita un método inflar(int n), que aumente en n unidades el aire del globo... Pero ¿quién tiene el aire que irá al globo? Por supuesto, el inflador. El total de aire (atributo del inflador) deberá disminuir en n, por ello la responsabilidad de inflar al globo (método para el inflador, cuya implementación podría modelarse como sigue en el gráfico), es del inflador.



En este caso, el Inflador, mediante un método público puede recibir un Globo y la cantidad de aire que necesita pasarle. Desde ese método, llamará al método paquete del Globo para inflar la cantidad de unidades necesarias. He ahí la verdadera responsabilidad: Una vez dado el aire al globo, él se sabe inflar. Sería incorrecto tanto que pueda ser inflado por cualquiera que no sea inflador, tanto como que fuera el inflador quien directamente modifique la variable cantidadAire del globo, ya que se violarían los principios de Responsabilidad y Encapsulamiento, sobre los que se hará hincapié reiteradas veces.

Relaciones entre clases

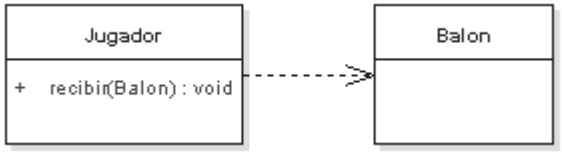
Por supuesto es necesaria y fundamental la interacción de las clases para que el modelado sea lo más próximo a la realidad, y, no solo eso, sino también para conseguir mayor rédito por el uso conjunto de las partes que por su empleo aislado (sinergia).

Una relación es una conexión entre elementos. Tenemos tres relaciones fundamentales en el Diseño Orientado a Objetos. Estas son la dependencia, la generalización y la agregación, un caso particular de la asociación.

Dependencia

Es una relación de uso, en la que, como su nombre lo indica, el cambio de una de las clases afecta a la otra, aunque no necesariamente también es a la inversa. Se utilizan generalmente cuando se quiere aclarar que un elemento se sirve de otro.

En la mayoría de los casos se utiliza esta relación para indicar que un objeto de determinada clase es, por ejemplo, atributo de un método de otra clase. Un jugador de fútbol, modelado como Jugador, puede recibir una pelota, modelada como Balon. Si un jugador recibe el balón, se podría decir que el jugador depende del balón para que su comportamiento sea uno u otro. Digamos: jugador.recibir(balon) por poner un ejemplo más visual.



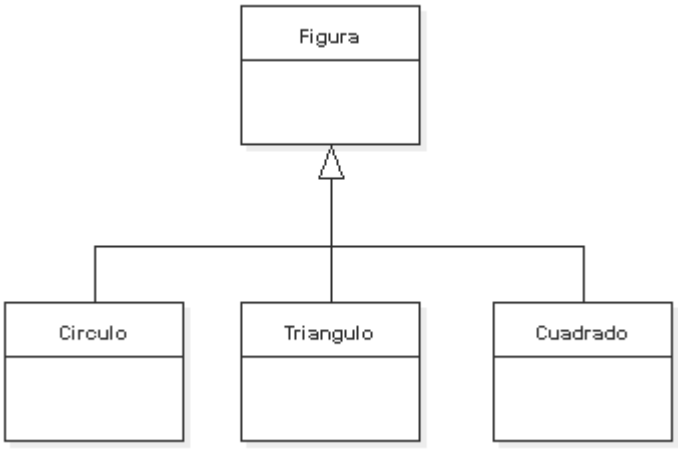
Generalización

Esta relación es la que existe entre un elemento general y un caso más específico de ese elemento. También llamada como la relación "es un", es la que caracteriza lo que conocemos y utilizamos como herencia.

En esta relación, las clases desde las que parten las flechas, son las versiones específicas de la que recibe la flecha.

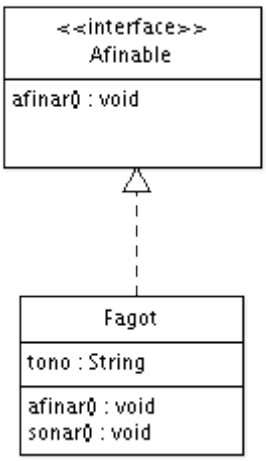
Se generaliza tanto hacia interfaces como a otras clases, abstractas o no.

El típico ejemplo de generalización es el de la clase Figura, que en sí no representa ninguna figura en particular, pero, al ser especializada, puede ser tanto un círculo como un triángulo, cuadrado...



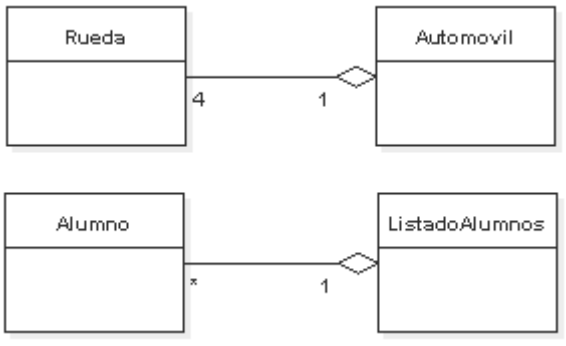
Por ejemplo (no está en el diagrama), toda figura posee un área. La relación de generalización puede servir para mostrar eso, poniendo un atributo area en la clase más general. Aún más, puede existir un método calcularArea(), cuya implementación variará según la figura que sea, pero que todas las figuras comparten. También se incluirá en la clase general Figura.

Para el caso que la generalización sea respecto a una interface, la relación tiene una pequeña diferencia, aquí un ejemplo:



Agregación
La relación conocida como parte/todo, o "es parte de" (leída al revés se dice "tiene"), es aquella que permite visualizar de modo más gráfico cómo se relacionan las clases de nuestro sistema, en forma sencilla. Mediante flechas que terminan en rombos, se designa cuál es la parte de qué otra cosa. El ejemplo más sencillo es el de un auto. Tiene un motor, ruedas... simplificando:

Cuando la relación de agregación puede tratarse de una cantidad mayor a uno, o variable, o quizás uno o ninguno... o todos los casos posibles, se agrega lo que se llama "rótulo de multiplicidad", el cual indica, a cada lado de la relación, la cantidad involucrada. Veamos dos ejemplos concretos:



En el primer caso, el automóvil está compuesto de cuatro ruedas (y otras cosas, pero estamos simplificando). Se muestra en los rótulos de multiplicidad, que UN automóvil está compuesto por CUATRO ruedas. No da lugar a dudas.
En el caso del Listado de Alumnos, UN listado está compuesto de VARIOS alumnos (no da certeza de número).

Convenciones más complejas se han adoptado para representar más casos, de las que sólo mostraremos algunas en esta tabla:

Rótulo	**Significado**

0..*	NINGUNA o VARIAS
1..*	UNA o VARIAS
0..n	NINGUNA a N
1..n	UNA a N

Bibliografía recomendada:

- El Lenguaje Unificado de Modelado, byBooch-Raumbaugh-Jacobson
- UML Distilled: A Brief Guide to the Standard Object Modeling Language, by Martin Fowler, Kendall Scott