UML: diagramas de clase con código

1. Relaciones entre clases

En una aplicación mínimamente compleja habrá varias clases cuyos métodos se llamarán unos a otros. Para que un método de la clase A pueda llamar a otro método de la clase B, la clase A debe poseer una referencia a un objeto de la clase B. Esta relación de posesión se representa con líneas que unen las distintas clases en el diagrama.

Las relaciones pueden ser varias:

- Asociación
- Agregación
- Composición
- Dependencia

También puede ocurrir que varias clases tengan operaciones o atributos en común y queramos abstraerlos utilizando mecanismos conocidos de los lenguajes orientados a objetos como la herencia y los interfaces. UML también permite representar estas relaciones entre clases mediante:

- Generalización (equivalente a la herencia en POO)
- Realización (equivalente a interfaces en Java)

2. Asociación

Es el tipo de relación más frecuente entre clases. Existe una relación de **asociación** entre ClaseA y ClaseB cuando en ClaseA hay un atributo de tipo ClaseB y/o viceversa.

Se representa como una línea continua, que puede estar o no acabada en una flecha en "V", según la **navegabilidad**. Además se suelen indicar los **roles** y la **multiplicidad**, que veremos un poco más abajo.

Navegabilidad

Si la flecha apunta de la ClaseA a la ClaseB, se lee como "ClaseA tiene una ClaseB" y se dice que la asociación o **navegabilidad** es **unidireccional**. Traducido a Java, esto significa que hay un atributo en la clase A que hace referencia a un objeto de la clase B.

Si no dibujamos la flecha, se lee "ClaseA tiene una ClaseB y ClaseB tiene una ClaseA", y se dice que la asociación o **navegabilidad** es **bidireccional**. En Java, ambas clases tendrían atributos que se hacen referencia recíprocamente.

Ejemplo: asociación bidireccional

Supongamos una aplicación de gestión con las clases Cliente y Pedido.

En el siguiente diagrama, Cliente guarda información sobre los pedidos y Pedido tiene información sobre el cliente que lo realizó. La navegabilidad es, por tanto, bidireccional:

Figura 3.1


```
public class Cliente {
        public Vector<Pedido> pedidos;
}
public class Pedido {
        public Cliente cliente;
}
```

El siguiente ejemplo muestra cómo se modelaría la asociación si la relación fuera a la inversa:

Los pedidos contienen información del cliente, pero los clientes no guardan pedidos:

Que traducido a Java sería:

```
public class Cliente {
        // esta vez el cliente NO guarda los pedidos
}
public class Pedido {
        public Cliente cliente;
}
```

Roles y multiplicidad

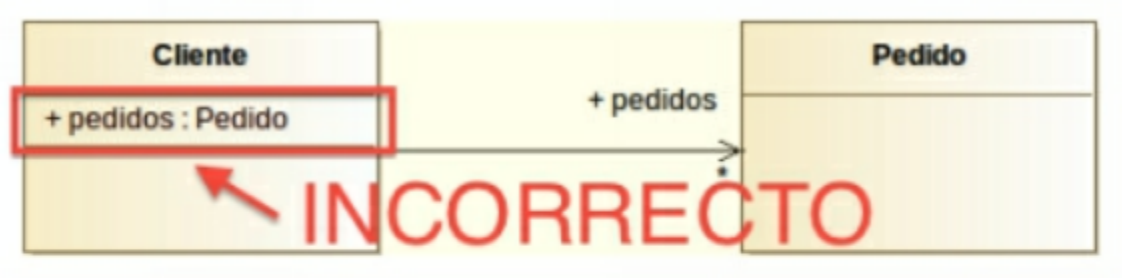
El **rol** o **papel** es el nombre de los atributos que intervienen en la relación

Figura 3.3

En el ejemplo anterior, se dice que "la clase Pedido asume el papel pedidos" en la asociación, lo que simplemente significa que el atributo de Cliente que hace referencia al vector de objetos Pedido se llamará "pedidos".

Es importante notar que el nombre de los atributos se escribe **en el lado contrario** a la clase que los contiene.

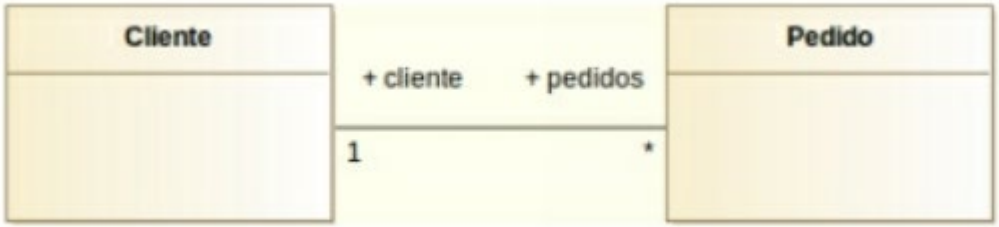
Un error común cuando representamos asociaciones es volver a incluir los atributos dentro de la clase. Lo siguiente sería **incorrecto**:



Además de los roles, como hemos visto en ejemplos anteriores, los extremos de la relación se pueden rotular con su **multiplicidad**, que indica cuántos objetos de una clase se pueden relacionar como **mínimo** y como **máximo** con objetos de la otra clase y se expresa del siguiente modo:

muliplicidad mínima .. multiplicidad máxima

Cuando las multiplicidades mínima y máxima son iguales, se suele representar con un único número. La multiplicidad de tipo "muchos" se representa con un asterisco: *. Por último, la multiplicidad 0..* (cero o más) se suele expresar como simplemente *.



- Un cliente se puede relacionar 0 o más pedidos (multiplicidad *, equivalente a 0..*) Un cliente se puede relacionar 0 o más pedidos (multiplicidad *, equivalente a 0..*)
- Un pedido debe tener un (y solo un) cliente (multiplicidad 1, equivalente a 1..1).

Ejemplo de multiplicidad: clientes y películas

Veamos en este otro ejemplo las posibilidades que aparecen según la multiplicidad que pongamos a la clase Pelicula:

Según el valor que le demos a "?":

- 1..1: Una instancia de cliente debe estar relacionada con exactamente una instancia de película, ni más ni menos.
- 1: Significa lo mismo que 1..1.
- 0..*: Un cliente puede no tener ninguna película alquilada, o puede tener varias (sin límite).
- *: Significa lo mismo que 0..*
- 0..3: Un cliente puede alquilar entre 0 y 3 películas.
- 2..3: Un cliente debe alquilar como mínimo 2 películas, pero como mucho puede alquilar 3

3. Agregación y composición

Según la referencia de UML [RJB05], la **agregación** y la **composición** son "asociaciones que representan una relación entre un todo y sus partes". Se puede leer como "ClaseB es un componente de ClaseA", o también "ClaseA está compuesto por ClaseB"

Las diferencias entre ellas son:

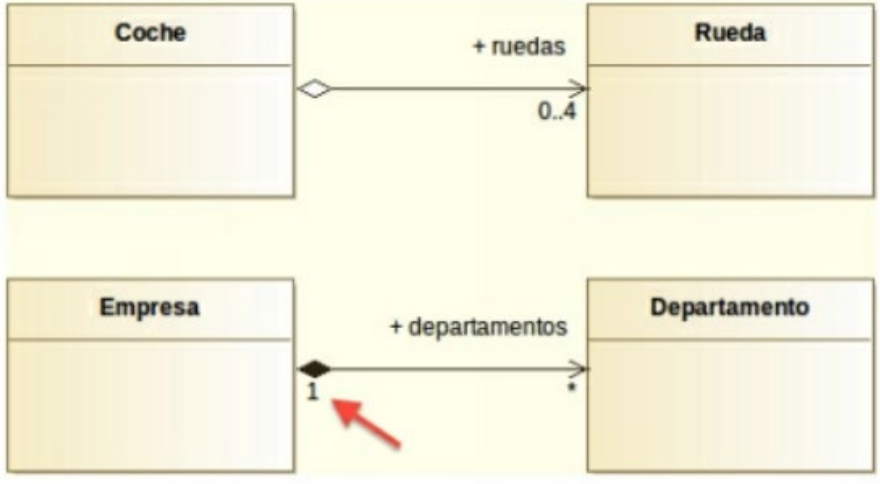
En la **agregación**, los objetos "parte" pueden seguir existiendo independientemente del objeto "todo".

En la **composición**, por el contrario, la vida de los objetos compuestos está íntimamente ligada a la del objeto que los compone, de manera que si el "todo" se destruye, las "partes" también se destruyen.

La agregación se representa uniendo las dos clases (todo / parte) con una línea continua y poniendo un rombo hueco en la clase "todo".

La composición es igual, pero con el rombo relleno.

Figura 3.6



La primera relación indica que el Coche está compuesto por Ruedas. Las ruedas pueden existir por sí mismas, por tanto la relación es de **agregación**.

En la segunda relación, una Empresa está compuesta por Departamentos. Los departamentos por sí mismos no pueden existir, ya que deben estar ligados a una empresa. Por este motivo, la relación ahora es de **composición**.

Una consecuencia de la composición es que una “parte” sólo puede relacionarse con 1 y sólo

1 “todo”, ya que si ese “todo” deja de existir, sus partes también deberán dejar de existir. Por tanto, la multiplicidad en la clase “todo” siempre será 1 en una composición.

Éste es un buen momento para explorar una útil función de Modelio: la auditoría. La auditoría consiste en verificar el modelo para asegurar que cumple con las normas de UML. Las infracciones se muestran en la vista Audit como errores o advertencias según su gravedad. Por ejemplo, si pusiéramos multiplicidad 1..* a la clase Empresa del ejemplo anterior, el programa nos advertiría con el siguiente error. Además, haciendo doble-clic encima nos daría información y pistas para solucionarlo.

La implementación en Java de la **agregación** es idéntica a la asociación:

```
public class Coche {
                private Rueda ruedas[4];
}
```

La **composición** se implementaría llamando a los destructores de las clases "parte" desde el destructor de la clase "todo" suponiendo que usemos un lenguaje con destructores, como C++:

```
class Todo
{
        Parte parte1; // construcción y destrucción implícitas
        Parte *parte2; // construcción y destrucción explícitas
public:
        Todo() {        parte2 = new Parte; }
        ~Todo() {
                delete parte2;
        }
}
```

En Java, sin embargo, la gestión de memoria está basada en la "recogida de basura" (garbage collection) y no existen los destructores, así que la **composición** se implementa exactamente igual que la agregación y la asociación:

```
public class Empresa {
        private Vector<Departamento> departamentos;
}
```

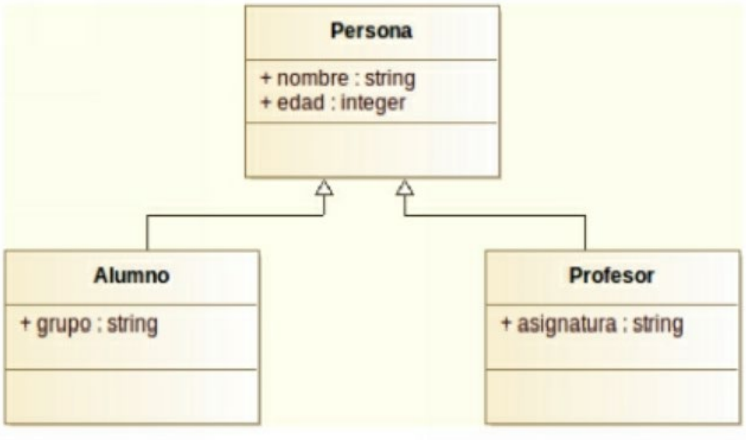
4. Generalización

La relación de **generalización** entre dos clases indica que una de las clases (la **subclase**) es una especialización de la otra (la **superclase**). Si ClaseA es la superclase y ClaseB la subclase, la generalización se podría leer como "ClaseB es una ClaseA" o "ClaseB es un tipo de ClaseA".

En Java (y la mayoría de lenguajes orientados a objetos), la generalización se conoce como **herencia**. La herencia se utiliza para abstraer en una superclase los métodos y/o atributos comunes a varias subclases.

A la subclase también se le puede llamar **clase hija** o **clase derivada**. A la superclase también se le puede llamar **clase padre** o **clase base**.

La generalización se expresa con una línea acabada en una flecha triangular hueca, dibujada desde la clase hija hacia la clase padre.



En Java:

```
public class Persona {
                    public String nombre; public int edad;
}
```

```
public class Alumno extends Persona {
                    public String grupo;
}


public class Profesor extends Persona {
                    public String asignatura;
}
```

6. Dependencia

Según el Manual de Referencia de UML [RJB05], una dependencia indica una relación semántica entre dos o más clases del modelo. La relación puede ser entre las clases propiamente dichas, es decir, no tiene por qué involucrar a instancias de las clases (como en la asociación, composición y agregación). Si ClaseA depende de ClaseB, significa que un cambio en ClaseB provoca un cambio en el significado de ClaseA.

Según esa definición, todas las relaciones vistas hasta ahora (asociación, agregación, composi- ción, generalización y realización) serían dependencias, pero por tener significados muy espe- cíficos se les ha dado nombres concretos. Por tanto, se podría decir que las dependencias son todas aquellas relaciones que no encajan en ninguna de las categorías anteriores.

En la práctica, y según coinciden muchos autores, la dependencia se usa cuando ClaseA utiliza brevemente a la ClaseB. Por "breve" se entiende normalmente lo que dura una llamada a un método.

Se representa como una línea punteada acabada en una flecha en "V" que va desde ClaseA a Clase B.

Concretando más aún, utilizaremos una relación de dependencia cuando una clase utilice un objeto de otra pero sin almacenarlo en ningún atributo (ya que entonces sería una asociación, agregación o composición). Ejemplos:

- Cuando a un método de la clase A se le pasa como parámetro un objeto de la clase B.
- Cuando un método de la clase A devuelve un objeto de la clase B
- Cuando un método de la clase A tiene una variable local de tipo B

Ejemplo: dependencia

Supongamos una clase Mapa capaz de obtener las coordenadas (latitud y longitud) a partir de una dirección postal y viceversa:

- La clase Coordenadas contiene los datos (latitud y longitud) de un punto geográfico.
- La clase Mapa tiene dos métodos:
- • El método direccionACoordenadas devuelve las coordenadas geográficas a partir de una dirección dada.
- • El método coordenadasADireccion devuelve la dirección más cercana a las coordenadas dadas.

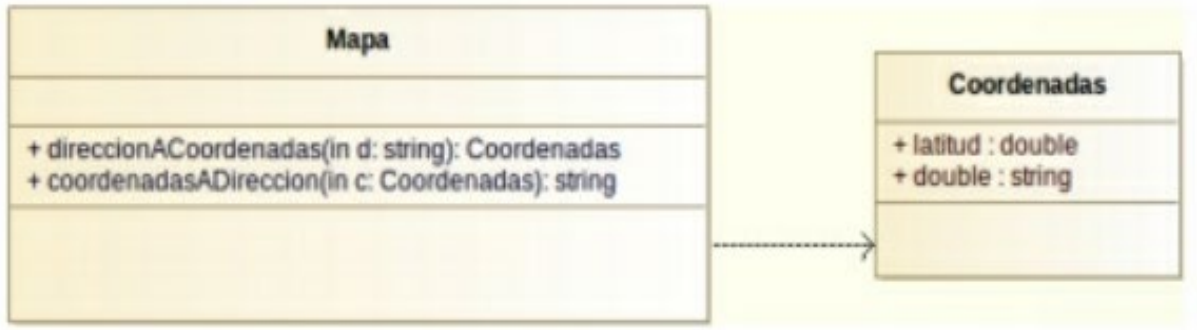
```
public class Mapa {
/** Devuelve coordenadas a partir de dirección postal
* @param d dirección postal
* @return coordenadas*/
                    public Coordenadas direccionACoordenadas(String d) {
                                        // ...
                    }
/** Devuelve dirección postal a partir de coordenadas
* @param c coordenadas
* @return dirección postal
*/
                    public String coordenadasADireccion(Coordenadas c) {
                    // ...
                    }
}
```

Coordenadas.java:

```
public class Coordenadas {
                      public double latitud;
                    public double longitud;
}
```

Ambos métodos utilizan brevemente un objeto de la clase Coordenadas, entendiendo por breve el hecho de que no la almacenan en ningún atributo de la clase. En consecuencia, Mapa **depende de** Coordenadas.

El diagrama que captura dicha relación sería:



7. Ejercicio guiado 1

En una empresa queremos guardar información de sus empleados y de los clientes con los siguientes requisitos:

- De los empleados queremos guardar su sueldo bruto.
- De los clientes queremos guardar su teléfono.
- Los clientes y los empleados tienen las siguientes características comunes: la edad y el nombre.
- Los directivos son un tipo de empleado. De ellos queremos conocer su categoría.
- Queremos una función que permita imprimir por pantalla todos los datos de cada persona.

Además existe una flota de coches de empresa a disposición de los directivos. Cada coche puede estar asignado a un único directivo y viceversa. De los vehículos necesitamos saber su matrícula y el kilometraje.