Relaciones entre objetos II: composición y agregación

Hasta aquí sabemos que, al momento de interactuar, los objetos generan relaciones que podemos clasificar en 3 categorías:

- Direccionalidad
- Cardinalidad
- Ordinalidad

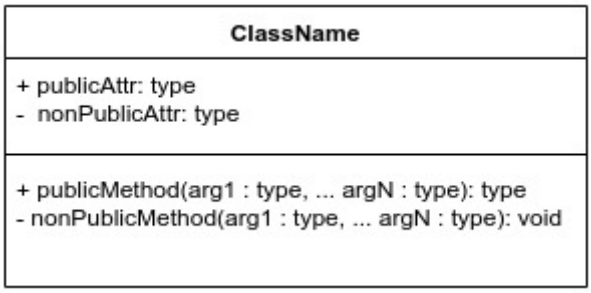
Ya analizamos las tres primeras, y lo hicimos tomando relaciones que llamamos "relaciones de agregación". Esa denominación no es casual, ya que nos queda una última característica para analizar las relaciones entre objetos que define **"relaciones de agregación" y "relaciones de composición".** Una relación es de composición o de agregación según la importancia o "el orden" de la relación. Antes de seguir, vamos a analizar los diagramas con los que mostramos las relaciones.

UML

Anteriormente, utilizamos algunos diagramas para mostrar las relaciones entre objetos. No son solo dibujos al pasar, sino que se trata de un lenguaje de modelado, específicamente diseñado para el modelado de software y sistemas orientados a objetos. Este lenguaje se denomina **lenguaje de modelado unificado o UML** por sus siglas en inglés. Si bien el lenguaje es bastante amplio (cubre relaciones entre clases, entre objetos, interacciones entre métodos, sistemas, componentes, máquinas de estados y una larga lista de etcéteras), aquí nos enfocaremos solo en los necesario e iremos ampliado las figuras y nomenclatura a medida que incorporemos nuevos conceptos. Por el momento, vamos a quedarnos con los diagramas de:

Clase

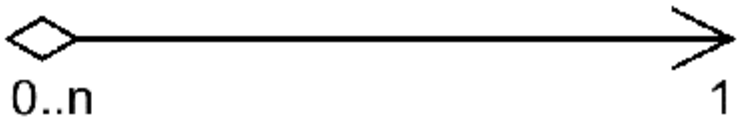
Diagramamos una clase de la siguiente manera:



Es un rectángulo con una sección dedicada al nombre de la clase en negrita. Luego, tiene una sección en la cual ponemos nuestros atributos. Cada atributo tiene un símbolo para indicar si es público (+) o si es cualquier otra cosa, **menos público** (-). Luego su nombre y separado por dos puntos (:) el tipo de dato. A continuación, una sección con los métodos, cada uno con el indicador de público o no público, el nombre del método, los parámetros y su tipo, y, al final, el tipo de dato del valor retornado o la palabra “void” si no retorna nada.

Relaciones

Hablamos en este artículo sobre relaciones entre objetos; para indicar relaciones tenemos:



Observamos una flecha para indicar la direccionalidad y los valores para indicar la cardinalidad. Pero ¿qué es ese pequeño rombo al principio de la flecha? A medida que vayamos necesitando más diagramas, los iremos introduciendo. Por ahora, para las clases y las relaciones entre objetos nos alcanza.

Orden de las relaciones

La ordinalidad denota la fortaleza de la relación. Así, podemos clasificar las relaciones en dos según su fuerza: las “relaciones de composición”, que son las más fuertes, y las “relaciones de agregación”, que son las más débiles.

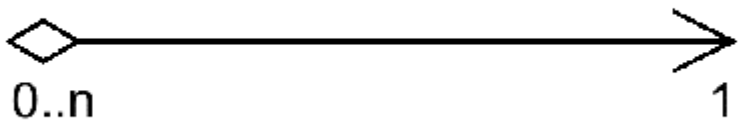
¿Qué queremos decir en realidad con que una relación es débil o fuerte? Las relaciones que vimos hasta ahora solían ser bastante ordinarias, por eso hablamos de relaciones de agregación. En las asociaciones libres, donde un objeto interactúa con otro, decimos que un objeto “colabora-con”. Por su lado, en las relaciones todo/parte (o parte/todo) hablamos de una relación de tipo “tiene-un”. Entonces, “Un auto ‘tiene-un’ motor”.

En una relación de agregación, la relación entre la parte y el todo es relativamente débil. En este escenario, parte y todo pueden existir por su lado sin verse afectadas. Tomemos como ejemplo el auto y el motor. Cuando se quita el motor del auto, para ser repararlo, este sigue existiendo. El mecánico lo pondrá sobre un banco y comenzará a trabajar en él. Quitarle el motor a un auto no le afecta en lo más mínimo. Por eso, podemos especificar más y decir solamente que “Un auto ‘usa-un’ motor”.

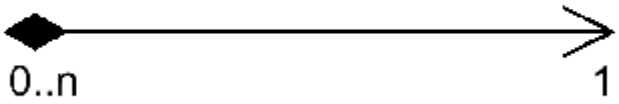
Por el contrario, en las relaciones de composición, la relación es muy fuerte; tan fuerte que, si la parte se separa del todo, esta deja de existir, se “esfuma”. Por ejemplo, tomemos el caso de un banco y las cuentas de las personas. La relación entre un banco y las cuentas es muy fuerte. Si el banco quebrara, o dejara de existir por cualquier motivo, las cuentas dejarían de existir. Las personas, aunque muy preocupadas y enojadas, seguirán existiendo. Las cuentas de un banco no tienen sentido sin el banco. Decimos entonces que el banco “es-dueño-de” sus cuentas, denotando la fortaleza en esa relación.

Entonces, volviendo a la pregunta que dejamos pendiente antes, ¿qué simboliza el rombo en la flecha del diagrama?

Indicamos una relación de **agregación** con:



Pero indicamos una relación de **composición** con:



Como siempre, la fortaleza de la relación está dada por el modelado que hagamos del problema. En nuestro proceso de abstracción decidiremos, caso por caso, cuál es la relación que corresponda. Volviendo al caso del auto, podríamos decir que un motor, por la razón que sea, no tiene ningún sentido por sí mismo, con lo cual podríamos modelarlo como una relación de composición. Asimismo, con el banco, si pensamos a las personas como clientes, entonces sí hacemos de eso una relación de composición también: si el banco no existe, entonces no existirán más los clientes.

A modo de resumen diremos, entonces, que estas relaciones parte/todo entran en la gran categoría de asociación y, según su orden, su fuerza, puede ser de agregación o de composición.

Código Java

Ambas relaciones pueden ser implementadas usando código Java. La relación más sencilla, la de agregación, es simplemente tener un objeto como atributo de otro objeto. Ya dijimos que son relaciones tipo “tiene-un”. Con lo cual, es algo que ya venimos haciendo:

```
public class Car {

  private Engine engine; //el auto “tiene-un” motor

  //getters & setters

}
```

```
public class Engine {

  private int displacement;
```

```
  private String manufacturer; // ya hacíamos agregación con Strings!

  //getters & setters

}
```

Por su lado, la relación de composición si bien es similar, tiene pequeña diferencia.

```
public class Car {

  private String brand;

  //el auto "crea" su motor, "es-dueño" de su motor

  private final Engine engine = new Engine();



  //getters & setters but not for engine attr

  public accelerate() {

    this.engine.moreRpm();

  }

}
```

De esta manera, el auto crea y usa su propio motor. Desde el momento en que hacemos en un "main" por ejemplo:

```
Car c = new Car();
```

El auto que guarda la variable c ya tiene una instancia de motor de la que es dueño. Si no usamos setters, evitamos que haya motores por fuera que le podamos dar al auto. Si no usamos getters, es porque queremos explicitar que el motor no sale del auto. No tiene razón de ser fuera del auto. Ahora bien, nada impide que nosotros hagamos, en cualquier parte del código, como en el main, por ejemplo, lo siguiente:

```
Engine e = new Engine();
```

Con lo cual tendríamos instancias vivas del motor sin estar dentro de un auto. Por eso, los más puristas dicen que hay que introducir una restricción adicional, haciendo de Engine una clase interna. Una clase interna es aquella que está definida dentro de otra. Algo así:

```
public class Car {

    private String brand;

    private final Engine engine = new Engine();

    public accelerate() {

      this.engine.moreRpm();

    }



    class Engine {

      int displacement;

      String manufacturer;

      void moreRpm() {

         //broooom

      }

    }

}
```

De esta forma sí queda formalmente explicitado que el motor solo vive dentro del auto y solo es usado por el auto. De hecho, no hicimos privados los atributos del motor, ya que se usarán solo en la clase Auto. Queda, entonces, **una relación muy fuerte a nivel de objetos y de clases.** De forma tal que si alguien quisiera crear un motor en un main no sería posible:

```
public class Test {

  public static void main(String[] args) {

    Car c = new Car();//ok

    Engine e = new Engine(); //compile error!

  }

}
```

Por supuesto que hay variantes y podemos complejizar aún más las cosas, pero está demostrado el punto. **La composición se refiere a relaciones muy fuertes entre la parte y el todo**.

Conclusiones

Como punto final solo podríamos agregar que todo este trabajo tiene sentido solo si es necesario para resolver el problema. Si somos muy estrictos podemos diseñar una relación de composición y plasmarla en código, aunque todo depende del problema que vayamos a resolver. ¿Tiene sentido lo que hicimos con el auto? Los mecánicos muchas veces trabajan con motores, los sacan de los autos, los arreglan o los mejoran y los vuelven a colocar. Mientras eso sucede, un auto no pierde su esencia, ¿o sí? De hecho, si tiene el capó cerrado, ni siquiera nos enteramos de que falta el motor. A la vista, ¡es un auto como cualquier otro!

De este modo, la grandísima mayoría del código que veamos implementará la relación más sencilla, la de agregación, por cuestiones de simplicidad y flexibilidad. Es más, seguramente utilicemos ambos términos como sinónimos, salvo en casos explícitos donde deberemos aclararlo.

¿Crees que las formas de relacionar objetos tienen un correlato correcto con la realidad?

¿Consideras que faltan más tipos de relaciones?

¿Crees que es necesaria la diferencia entre agregación y composición? Te invito a reflexionar sobre estos interrogantes.