Relaciones entre clases Python

Ricardo Pérez López

IES Doñana, curso 2020/2021

Índice general

1. Relaciones básicas

1.1. Introducción

Los objetos de un programa interactúan entre sí durante la ejecución del mismo, por lo que decimos que **los objetos se relacionan entre sí**.

Las **relaciones entre objetos** pueden ser de varios tipos.

Por ejemplo, cuando un objeto **envía un mensaje** a otro, tenemos un claro ejemplo de relación del tipo **usa** (el primer objeto «usa» al segundo).

Otras veces, los objetos **contienen** a otros objetos, o bien **forman parte** de otros objetos.

Finalmente, a veces las relaciones entre los objetos son meramente **conceptuales**:

- Son relaciones que **no se reflejan** directamente **en el código fuente** del programa, sino que afloran durante el **análisis** del problema a resolver o como parte del **diseño** de la solución.
- Es decir: aparecen durante las etapas de análisis o diseño del sistema y se representan, por tanto, en los **documentos de análisis y diseño**.

Cuando una o varias instancias de una clase está relacionada con una o varias instancias de otra clase, podemos decir que ambas clases también están relacionadas.

Una **relación entre clases** representa un conjunto de posibles relaciones entre instancias de esas clases.

La **multiplicidad de una clase en una relación** representa la cantidad de instancias de esa clase que se pueden relacionar con una instancia de la otra clase en esa relación.

El **lenguaje UML** describe la sintaxis y la semántica de las posibles *multiplicidades* que se pueden usar en una relación entre clases, así como los distintos *tipos de relaciones entre clases* que se pueden dar en un sistema orientado a objetos.

En el módulo *Entornos de desarrollo* se estudian con detalle tanto el lenguaje UML como los distintos tipos de relaciones que se pueden establecer entre clases.

En *Programación* sólo trabajaremos con las relaciones que se reflejen en el código fuente del programa y que, por tanto, formen parte del mismo.

Por tanto, las relaciones conceptuales que se puedan establecer a nivel semántico durante el análisis o el diseño del sistema no se verán aquí y sólo se verán en *Entornos de desarrollo*.

1.2. Asociación

Una **asociación** simple es una relación genérica que se establece entre dos clases.

En Programación se usa principalmente para representa el hecho de que una clase «usa» a la otra de alguna forma.

Normalmente se da cuando un método de una clase necesita acceder a una instancia de otra clase.

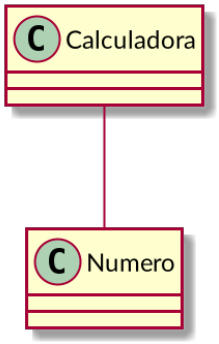
Esa instancia la puede recibir como argumento, o bien puede crearla y destruirla el propio método.

Por ejemplo:

```
class Calculadora:
    @staticmethod
    def suma(x, y):
        """Devuelve la suma de dos instancias de la clase Numero."""
        return x.get_valor() + y.get_valor()
```

Aquí se establece una *asociación* entre las clases `Calculadora` y `Numero`.

En UML, podríamos representarla así:



1.3. Agregación

La **agregación** es una relación que se establece entre una clase (la **agregadora**) y otra clase (la **agregada**).

Representa la relación «**tiene**»: la agregadora *tiene* a la agregada.

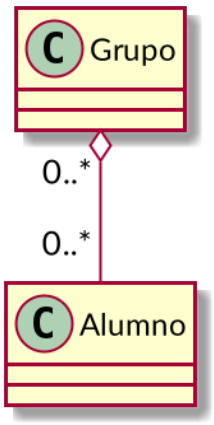
Podríamos decir que la agregada **forma parte** de la agregadora, pero de una forma **débil**, ya que los objetos de la clase agregadora y de la clase agregada tienen su existencia propia, independiente.

Por tanto:

- La agregada puede formar parte de varias agregadoras.
- Según sea el caso, el objeto de la clase agregada puede existir aunque no forme parte de ningún objeto agregador.
- La clase agregadora no tiene por qué ser la responsable de crear el objeto agregado.
- Cuando se destruye un objeto de la clase agregadora, no es necesario destruir los objetos de la clase agregada.

Por ejemplo:

- Los grupos tienen alumnos. Un alumno puede pertenecer a varios grupos, y un alumno existe por sí mismo aunque no pertenezca a ningún grupo.
- La clase `Grupo` «agrega» a la clase `Alumno`.




```
class Grupo:
    def __init__(self):
        self.__alumnos = []  # Guarda una lista de referencias a Alumnos

    def get_alumnos(self):
        return self.__alumnos

    def meter_alumno(self, alumno):
        self.__alumnos.append(alumno)

    def sacar_alumno(self, alumno):
        try:
            self.__alumnos.remove(alumno)
        except ValueError:
            raise ValueError("El alumno no está en el grupo")

daw1 = Grupo()           # Los objetos los crea...
pepe = Alumno()          # ... el programa principal, así que ...
juan = Alumno()          # ... ningún objeto crea a otro.
daw1.meter_alumno(pepe)  # Metemos en __alumnos una referencia a pepe
daw1.meter_alumno(juan)  # Metemos en __alumnos una referencia a juan
daw1.sacar_alumno(pepe)  # Eliminamos de __alumnos la referencia a pepe
daw2 = Grupo()           # Se crea otro grupo
daw2.meter_alumno(juan)  # juan está en daw1 y daw2 al mismo tiempo
```

1.4. Composición

2. Herencia

2.1. Concepto de herencia

2.2. Modos

2.2.1. Simple

2.2.2. Múltiple

2.3. Superclases y subclases