CURSO DE PROGRAMACIÓN FULL STACK

RELACIONES ENTRE CLASES

PARADIGMA ORIENTADO A OBJETOS

GUÍA RELACIONES ENTRE CLASES

RELACIONES ENTRE CLASES

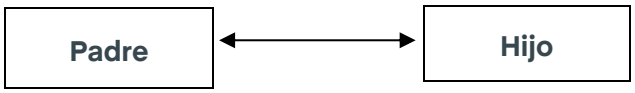
Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las clases no actúan aisladas entre sí, al contrario las clases están relacionadas unas con otras. Una clase puede ser un tipo de otra clase —generalización— o bien puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación entre las dos clases.

En la programación orientada a objetos, un objeto se comunica con otro objeto para utilizar la funcionalidad y los servicios proporcionados por ese objeto. Por ejemplo: un objeto curso tiene varios objetos alumnos y un objeto profesor.
Esto significa que, gracias a la relación entre objetos, el objeto curso puede tener toda la información que necesita.

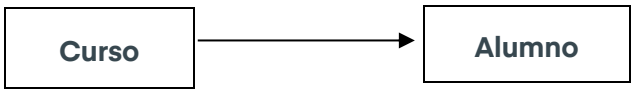
La relación entre dos clases separadas se establece a través de sus Objetos. Es decir, las clases se conectan juntas conceptualmente. Las relaciones entre clases realmente significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo.

La relación mas simple es la asociación, esta relación es entre dos objetos como habíamos dicho previamente. En las relaciones de asociación se puede establecer una relación bidireccional, que los objetos que están al extremo de una relación pueden "conocerse "entre sí, o una relación unidireccional que solamente uno de ellos "conoce" a otro.

Bidireccional:



Unidireccional:



Dentro de la asociación simple existe la composición y la agregación, que son las dos formas de relaciones entre clases.

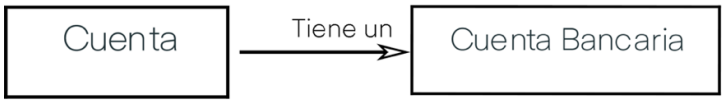
AGREGACIÓN

Representa un tipo de relación muy particular, en la que una clase es instanciada por otro objeto y clase. La agregación se podría definir como el momento en que dos objetos se unen para trabajar juntos y así, alcanzar una meta. Un punto a tomar muy en cuenta es que ambos objetos son independientes entre sí. Para validar la agregación, la frase **"Usa un"** debe tener sentido, por ejemplo: El programador usa una computadora. El objeto computadora va a existir más allá de que exista o no el objeto programador.

En agregación, ambos objetos pueden sobrevivir individualmente, lo que significa que al borrar un objeto no afectará a la otra entidad.

COMPOSICIÓN

La composición es una relación más fuerte que la agregación, es una "relación de vida", es decir, que el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye. La composición es un tipo de relación dependiente en dónde un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase **"Tiene un",** debe tener sentido, por ejemplo: el cliente tiene una cuenta bancaria. Esta relación es una composición, debido a que al eliminar el cliente la cuenta bancaria no tiene sentido, y también se debe eliminar, es decir, la cuenta existe sólo mientras exista el cliente.



RELACIONES EN CÓDIGO

Recordemos que las relaciones entre clases significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo. Pero a la hora de poner un atributo en una clase, debemos ver el tipo de relación de esas clases.

En las relaciones, tanto composición como agregación, las relaciones pueden ser de **uno a uno**, de **uno a muchos**, de **muchos a uno**, de **muchos a muchos**. El tipo de relación se ve representada a la hora de poner el objeto como forma de atributo en la clase que recibe la relación. Por ahora vamos a trabajar solo con **uno a uno** y **uno a muchos** porque son las que podemos representar en código por ahora.

Uno a uno

Por cada objeto tenemos una relación con un solo objeto. Ejemplo: para **un** curso tengo **un** profesor. En código se representa con un atributo que sea un objeto.

```
Public class Profesor {

private String nombre;
private String apellido;

}
```

```
Public class Curso {

private Profesor profesor;

public Profesor getProfesor() {
return profesor;
}

public void setProfesor(Profesor profesor){
this.profesor = profesor;
}
```

En este ejemplo en el Main vamos a tener que crear un objeto Profesor, para poder guardarlo en el Curso. Para guardar el objeto podemos usar el set que se va a generar de dicho objeto Profesor, ya que es un atributo de la clase Curso.

Main

```
Profesor profesor = new Profesor();
profesor.setNombre("Agustín");
profesor.setApellido("Lima");
Curso curso = new Curso();
curso.setProfesor(profesor);
```

Uno a muchos

Por cada objeto tenemos una relación con muchos objetos de una clase. Ejemplo: para un curso tengo muchos alumnos. En java para guardar varios objetos de una clase utilizamos colecciones. Y como las listas son las colecciones más rápidas de llenar, utilizamos una lista

```
Public class Alumno {

private String nombre;
private String apellido;

}
```

```
Public class Curso {

private List<Alumno> alumnos;

public List<Alumno> getAlumnos() {
return alumnos;
}

public void setAlumno(sList<Alumno> alumnos){
this.alumnos = alumnos;
}
```

En este ejemplo en el Main vamos a tener que crear varios objetos Alumno para después guardarlos en un ArrayList de tipo Alumno, para poder guardarlo en el Curso. Para guardar el objeto podemos usar el set que se va a generar de dicho ArrayList de tipo Alumno, ya que es un atributo de la clase Curso.

Main

```
Alumno alumno1 = new Alumno();
alumno1.setNombre("Mariela");
alumno1.setApellido("Gadea");
ArrayList<Alumno> alumnos = new ArrayList();
alumnos.add(alumno1);
Curso curso = new Curso();
curso.setAlumnos(alumnos);
```

UML

El lenguaje de modelado unificado (UML) es un lenguaje de modelado de propósito general. El objetivo principal de UML es definir una forma estándar de visualizar la forma en que se ha diseñado un sistema mediante diagramas. Es bastante similar a los planos utilizados en otros campos de la ingeniería.

UML no es un lenguaje de programación, es más bien un lenguaje visual. Usamos diagramas UML para representar el comportamiento y la estructura de un sistema. Estos diagramas se hacen siempre previos a la codificación del programa en sí, también para facilitar después la creación del programa si ya tenemos en claro que debemos crear. Existen varios tipos de diagramas de programación que podemos hacer con UML, en el que vamos a hacer hincapié nosotros es el diagrama de clases.

DIAGRAMAS DE CLASES

Es el componente básico de todos los programas orientados a objetos. Usamos diagramas de clases para representar la estructura de un sistema mostrando las clases del sistema, sus métodos y atributos. Los diagramas de clases también nos ayudan a identificar la relación entre diferentes clases u objetos. Ya se relaciones entre clases o herencia.
Cada clase está representada por un rectángulo que tiene una subdivisión de tres compartimentos: nombre, atributos y métodos.

Hay tres tipos de modificadores que se utilizan para decidir la visibilidad de atributos y métodos:

\+ se usa para el modificador de acceso public.
\# se usa para el modificador de acceso protected.
\- se usa para el modificador de acceso private.

Libro
-String titulo -Integer ejemplares
+void préstamo() +void devolución()

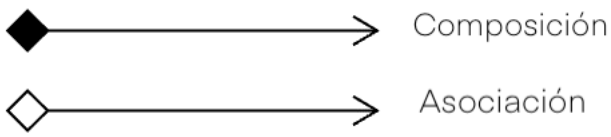
Si tuviéramos una interfaz, se vería así

<<interface>> Interfaz
+void método()

Nota: El concepto de interfaz se desarrollará en la siguiente guía.

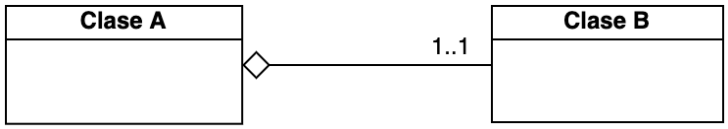
Relaciones entre clases

Las relaciones entre clases se representan con flechas entre las clases. La clase que recibe la relación de la otra clase, como un objeto de la otra clase, es la clase a la que lo toca el rombo.



Y para representar el tipo de relación, ya sea **uno a uno**, de **uno a muchos**, de **muchos a uno** o **muchos a muchos,** es con un símbolo para cada relación en la flecha.
A continuación, veremos uno de los muchos ejemplos:

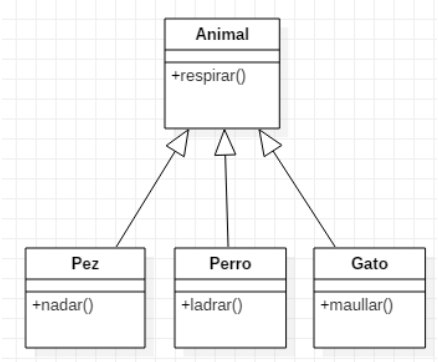
1...1	Uno a uno
1...*	Uno a muchos o muchos a uno



En el primer ejemplo la Clase A tiene a la clase B por ser una composición y en el segundo ejemplo, la clase A usa a la clase B por ser una asociación.

Herencia

El concepto de herencia se desarrollará en la siguiente guía.
La herencia se representa con la siguiente flecha:



PREGUNTAS DE APRENDIZAJE

1) La relación más fuerte es la

   a) Agregación
   b) Composición
   c) Dependencia
   d) Ninguna de las anteriores

2) La relación más débil es la

   a) Agregación
   b) Composición
   c) Dependencia
   d) Ninguna de las anteriores

3) La relación es entre

   a) Clases
   b) Interfaces
   c) Métodos
   d) Ninguna de las anteriores

4) La relación se representa en una clase con un

   a) Método
   b) Constructor
   c) Atributo objeto de esa clase
   d) Ninguna de las anteriores

5) El modificador de acceso public en los UML se representa con un

   a) Menos (-)
   b) Más (+)
   c) Numeral (#)
   d) Ninguna de las anteriores

6) El modificador de acceso private en los UML se representa con un

   a) Menos (-)
   b) Más (+)
   c) Numeral (#)
   e) Ninguna de las anteriores