CURSO DE PROGRAMACIÓN FULL STACK

RELACIONES ENTRE CLASES

PARADIGMA ORIENTADO A OBJETOS

GUÍA RELACIONES ENTRE CLASES

RELACIONES ENTRE CLASES

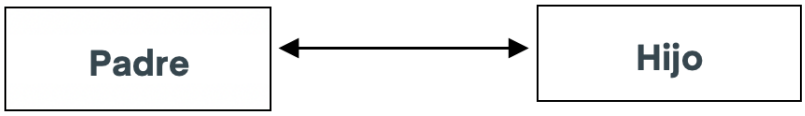
Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las clases no actúan aisladas entre sí, al contrario las clases están relacionadas unas con otras. Una clase puede ser un tipo de otra clase —generalización— 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. Además, nos ayuda para la reutilización de código, ya que podemos usar todo lo de la clase que ya habíamos escrito,

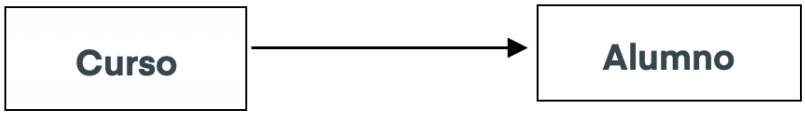
La relación entre dos clases separadas se establece a través de sus Objetos. Es decir, las clases se conectan juntas conceptualmente. Las relaciones entre clases realmente significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo.

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

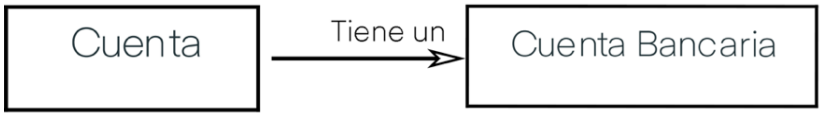
Representa un tipo de relación muy particular, en la que una clase es instanciada por otro objeto y clase. La agregación se podría definir como el momento en que dos objetos se unen para trabajar juntos y así, alcanzar una meta. Un punto a tomar muy en cuenta es que ambos objetos son independientes entre 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 **cero a uno**, de **uno a muchos** y de **cero 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 más comunes.

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.

```java
public class Profesor {

private String nombre;
private String apellido;


}
```

```java
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

```java
Profesor profesor = new Profesor();
profesor.setNombre("Agustín");
profesor.setApellido("Lima");
Curso curso = new Curso();
curso.setProfesor(profesor); // Seteamos un profesor en el Curso
```

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

```java
Public class Alumno {

private String nombre;
private String apellido;

}
```

```java
Public class Curso {

private List<Alumno> alumnos;

public List<Alumno> getAlumnos() {
return alumnos;
}
public void setAlumno(List<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); // Agregamos el alummno a la lista
Curso curso = new Curso();
curso.setAlumnos(alumnos); // Seteamos la lista de alumnos en el Curso
```

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()

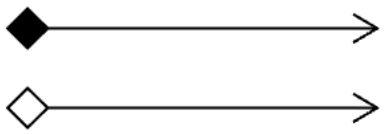
Si tuviéramos una interfaz, se vería así

<<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.

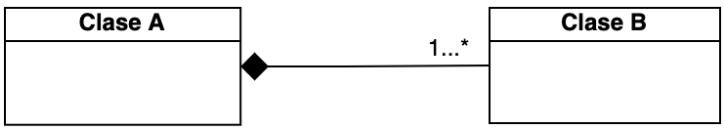


Composición

Agregación

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

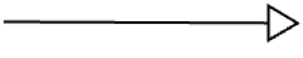
1...1	Uno a uno
1...*	Uno a muchos o muchos a uno



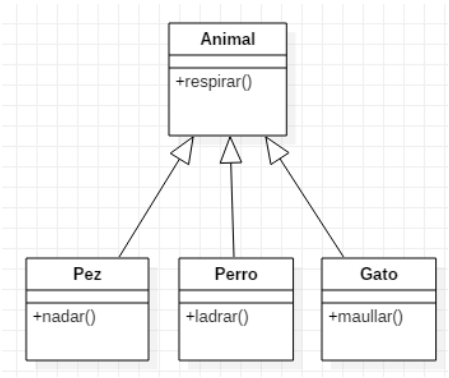
En el primer ejemplo la Clase A tiene a la clase B por ser una composición y en el segundo ejemplo, la clase A usa a la clase B por ser una agregación.

HERENCIA

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



Herencia



EJERCICIOS DE APRENDIZAJE

En este módulo de POO, vamos a empezar a ver cómo dos o más clases pueden relacionarse entre sí, ya sea por una relación entre clases o mediante una herencia de clases.

Nota: instalar el easyUML por si queremos ver nuestros proyectos como diagramas UML. Encontrarán como instalarlo en Moodle.

VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Realizar un programa para que una Persona pueda adoptar un Perro. Vamos a contar de dos clases. Perro, que tendrá como atributos: nombre, raza, edad y tamaño; y la clase Persona con atributos: nombre, apellido, edad, documento y Perro.
   Ahora deberemos en el main crear dos Personas y dos Perros. Después, vamos a tener que pensar la lógica necesaria para asignarle a cada Persona un Perro y por ultimo, mostrar desde la clase Persona, la información del Perro y de la Persona.

2. Realizar el juego de la ruleta rusa de agua en Java. Como muchos saben, el juego se trata de un número de jugadores, que, con un revolver de agua, el cual posee una sola carga de agua, se dispara y se moja. Las clases a hacer del juego son las siguientes:

   Clase Revolver de agua: esta clase posee los siguientes atributos: posición actual (posición del tambor que se dispara, puede que esté el agua o no) y posición agua (la posición del tambor donde se encuentra el agua). Estas dos posiciones, se generarán aleatoriamente.

   Métodos:

   - llenarRevolver(): le pone los valores de posición actual y de posición del agua. Los valores deben ser aleatorios.
   - mojar(): devuelve true si la posición del agua coincide con la posición actual
   - siguienteChorro(): cambia a la siguiente posición del tambor
   - toString(): muestra información del revolver (posición actual y donde está el agua)

   Clase Jugador: esta clase posee los siguientes atributos: id (representa el número del jugador), nombre (Empezara con Jugador más su ID, "Jugador 1" por ejemplo) y mojado (indica si está mojado o no el jugador). El número de jugadores será decidido por el usuario, pero debe ser entre 1 y 6. Si no está en este rango, por defecto será 6.

   Métodos:

- disparo(Revolver r): el método, recibe el revolver de agua y llama a los métodos de mojar() y siguienteChorro() de Revolver. El jugador se apunta, aprieta el gatillo y si el revolver tira el agua, el jugador se moja. El atributo mojado pasa a false y el método devuelve true, sino false.

Clase Juego: esta clase posee los siguientes atributos: Jugadores (conjunto de Jugadores) y Revolver

Métodos:

- llenarJuego(ArrayList<Jugador>jugadores, Revolver r): este método recibe los jugadores y el revolver para guardarlos en los atributos del juego.
- ronda(): cada ronda consiste en un jugador que se apunta con el revolver de agua y aprieta el gatillo. Sí el revolver tira el agua el jugador se moja y se termina el juego, sino se moja, se pasa al siguiente jugador hasta que uno se moje. Si o si alguien se tiene que mojar. Al final del juego, se debe mostrar que jugador se mojó.
  Pensar la lógica necesaria para realizar esto, usando los atributos de la clase Juego.

3. Realizar una baraja de cartas españolas orientada a objetos. Una carta tiene un número entre 1 y 12 (el 8 y el 9 no los incluimos) y un palo (espadas, bastos, oros y copas). Esta clase debe contener un método toString() que retorne el número de carta y el palo. La baraja estará compuesta por un conjunto de cartas, 40 exactamente.

Las operaciones que podrá realizar la baraja son:

- barajar(): cambia de posición todas las cartas aleatoriamente.
- siguienteCarta(): devuelve la siguiente carta que está en la baraja, cuando no haya más o se haya llegado al final, se indica al usuario que no hay más cartas.
- cartasDisponibles(): indica el número de cartas que aún se puede repartir.
- darCartas(): dado un número de cartas que nos pidan, le devolveremos ese número de cartas. En caso de que haya menos cartas que las pedidas, no devolveremos nada, pero debemos indicárselo al usuario.
- cartasMonton(): mostramos aquellas cartas que ya han salido, si no ha salido ninguna indicárselo al usuario
- mostrarBaraja(): muestra todas las cartas hasta el final. Es decir, si se saca una carta y luego se llama al método, este no mostrara esa primera carta.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Ahora se debe realizar unas mejoras al ejercicio de Perro y Persona. Nuestro programa va a tener que contar con muchas personas y muchos perros. El programa deberá preguntarle a cada persona, que perro según su nombre, quiere adoptar. Dos personas no pueden adoptar al mismo perro, si la persona eligió un perro que ya estaba adoptado, se le debe informar a la persona.
   Una vez que la Persona elige el Perro se le asigna, al final deberemos mostrar todas las personas con sus respectivos perros.

2. Nos piden hacer un programa sobre un Cine, que tiene una sala con un conjunto de asientos (8 filas por 6 columnas). De Cine nos interesa conocer la película que se está reproduciendo, la sala con los espectadores y el precio de la entrada. Luego, de las películas nos interesa saber el título, duración, edad mínima y director. Por último, del espectador, nos interesa saber su nombre, edad y el dinero que tiene disponible.
   Para representar la sala con los espectadores vamos a utilizar una matriz. Los asientos son etiquetados por una letra y un número la fila A1 empieza al final del mapa como se muestra en la tabla. También deberemos saber si el asiento está ocupado por un espectador o no, si esta ocupado se muestra una X, sino un espacio vacío.

8A	X | 8B	X | 8C	X | 8D	| 8E	X | 8F X
7A	X | 7B	X | 7C	X | 7D	X | 7E	| 7F X
6A	| 6B	X | 6C	| 6D	X | 6E	X | 6F
5A	X | 5B	| 5C	X | 5D	X | 5E	X | 5F X
4A	X | 4B	X | 4C	X | 4D	X | 4E	X | 4F X
3A	| 3B	X | 3C	X | 3D	| 3E	X | 3F X
2A	X | 2B	| 2C	X | 2D	X | 2E	X | 2F
1A	X | 1B	X | 1C	X | 1D	X | 1E	X | 1F X

   Se debe realizar una pequeña simulación, en la que se generen muchos espectadores y se los ubique en los asientos aleatoriamente (no se puede ubicar un espectador donde ya este ocupado el asiento).
   Los espectadores serán ubicados de uno en uno y para ubicarlos tener en cuenta que sólo se podrá sentar a un espectador si tiene el dinero suficiente para pagar la entrada, si hay espacio libre en la sala y si tiene la edad requerida para ver la película. En caso de que el asiento este ocupado se le debe buscar uno libre.

Al final del programa deberemos mostrar la tabla, podemos mostrarla con la letra y numero de cada asiento o solo las X y espacios vacíos.

3. Ha llegado el momento de poner de prueba tus conocimientos. Para te vamos a contar que te ha contratado "La Tercera Seguros", una empresa aseguradora que brinda a sus clientes coberturas integrales para vehículos.

Luego de un pequeño relevamiento, te vamos a pasar en limpio los requerimientos del sistema que quiere realizar la empresa.

a. Gestión Integral de clientes. En este módulo vamos a registrar la información personal de cada cliente que posea pólizas en nuestra empresa. Nombre y apellido, documento, mail, domicilio, teléfono.
b. Gestión de vehículos. Se registra la información de cada vehículo asegurado. Marca, modelo, año, número de motor, chasis, color, tipo (camioneta, sedán, etc.).
c. Gestión de Pólizas: Se registrará una póliza, donde se guardará los datos tanto de un vehículo, como los datos de un solo cliente. Los datos incluidos en ella son: número de póliza, fecha de inicio y fin de la póliza, cantidad de cuotas, forma de pago, monto total asegurado, incluye granizo, monto máximo granizo, tipo de cobertura (total, contra terceros, etc.). Nota: prestar atención al principio de este enunciado y pensar en las relaciones entre clases. Recuerden que pueden ser de uno a uno, de uno a muchos, de muchos a uno o de muchos a muchos.
d. Gestión de cuotas: Se registrarán y podrán consultar las cuotas generadas en cada póliza. Esas cuotas van a contener la siguiente información: número de cuota, monto total de la cuota, si está o no pagada, fecha de vencimiento, forma de pago (efectivo, transferencia, etc.).

Debemos realizar el diagrama de clases completo, teniendo en cuenta todos los requerimientos arriba descriptos. Modelando clases con atributos y sus correspondientes relaciones. Para hacer el diagrama podes utilizar easyUml de Java o utilizar una pagina que nos permite hacer diagramas online: diagramas online.

4. Desarrollar un simulador del sistema de votación de facilitadores en Egg-

El sistema de votación de Egg tiene una clase llamada Alumno con los siguientes atributos: nombre completo, DNI y cantidad de votos. Además, crearemos una clase Simulador que va a tener los métodos para manejar los alumnos y sus votaciones. Estos métodos serán llamados desde el main.

- La clase Simulador debe tener un método que genere un listado de alumnos manera aleatoria y lo retorne. Las combinaciones de nombre y apellido deben ser generadas de manera aleatoria. Nota: usar listas de tipo String para generar los nombres y los apellidos.

- Ahora hacer un generador de combinaciones de DNI posibles, deben estar dentro de un rango real de números de documentos. Y agregar a los alumnos su DNI. Este método debe retornar la lista de dnis.

- Ahora tendremos un método que, usando las dos listas generadas, cree una cantidad de objetos Alumno, elegidos por el usuario, y le asigne los nombres y los dnis de las dos listas a cada objeto Alumno. No puede haber dos alumnos con el mismo dni, pero si con el mismo nombre.

- Se debe imprimir por pantalla el listado de alumnos.

- Una vez hecho esto debemos generar una clase Voto, esta clase tendrá como atributos, un objeto Alumno que será el alumno que vota y una lista de los alumnos a los que votó.

- Crearemos un método votación en la clase Simulador que, recibe el listado de alumnos y para cada alumno genera tres votos de manera aleatoria. En este método debemos guardar a el alumno que vota, a los alumnos a los que votó y sumarle uno a la cantidad de votos a cada alumno que reciba un voto, que es un atributo de la clase Alumno.
  Tener en cuenta que un alumno no puede votarse a sí mismo o votar más de una vez al mismo alumno. Utilizar un hashset para resolver esto.

- Se debe crear un método que muestre a cada Alumno con su cantidad de votos y cuales fueron sus 3 votos.

- Se debe crear un método que haga el recuento de votos, este recibe la lista de Alumnos y comienza a hacer el recuento de votos.

- Se deben crear 5 facilitadores con los 5 primeros alumnos votados y se deben crear 5 facilitadores suplentes con los 5 segundos alumnos más votados. A continuación, mostrar los 5 facilitadores y los 5 facilitadores suplentes.