UML

Unified Modeling Language

Rapida explicacion y ejemplos sobre UML.

Programación con Objetos 2

Universidad Nacional de Quilmes

Clase

La forma de definir una clase en UML es a través de una caja separada en tres partes.

En la primer parte de la caja se escribe el nombre de la clase en cuestión.

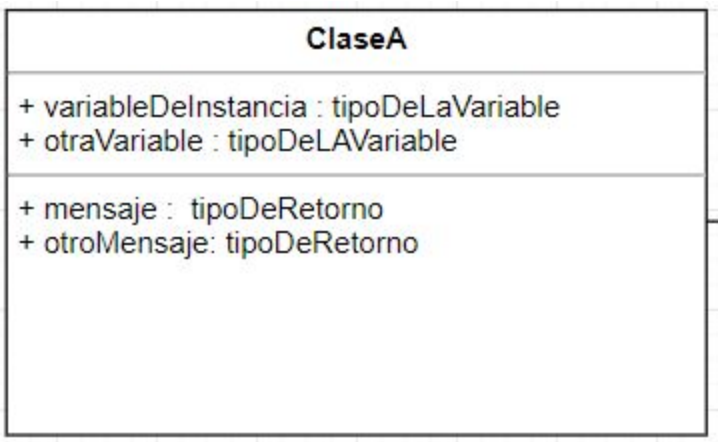
En la segunda parte se escriben las variables de instancia (o colaboradores internos) de la clase cuyos tipos son **SOLAMENTE** tipos primitivos o tipos básicos provistos por el lenguaje.

En la tercer parte se escriben los mensajes que entiende este objeto.

Constructor: Identificamos al constructor como un mensaje más dentro del protocolo de la clase, pero subrayado.

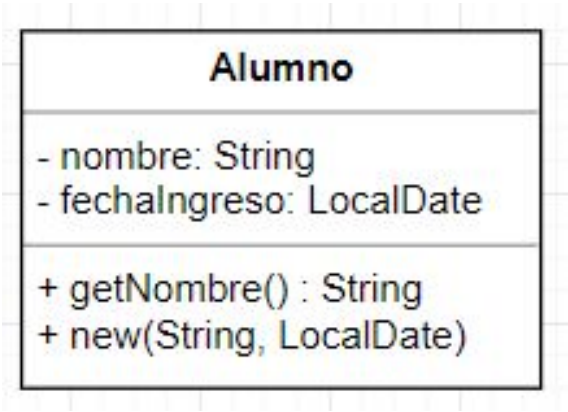
Tipos: Luego de escribir el nombre de la variable y/o el nombre del mensaje, se debe escribir el tipo de la variable o el tipo de retorno del mensaje.

Visibilidad: Para definir la visibilidad de un mensaje o variable se utilizan los símbolos **más (+) y menos (-)** para identificar visibilidad pública y privada respectivamente.



Veamoslo con un ejemplo.

Conocemos una alumno universitarios de la cual conocemos su nombre y la fecha en que ingresó a la facultad. Por el momento solo nos interesa conocer el nombre del alumno. Definimos entonces la clase Alumno. Notar que las variables *fechaIngreso* y *nombre* son variables **privadas**



Y ¿Como se ve el código?

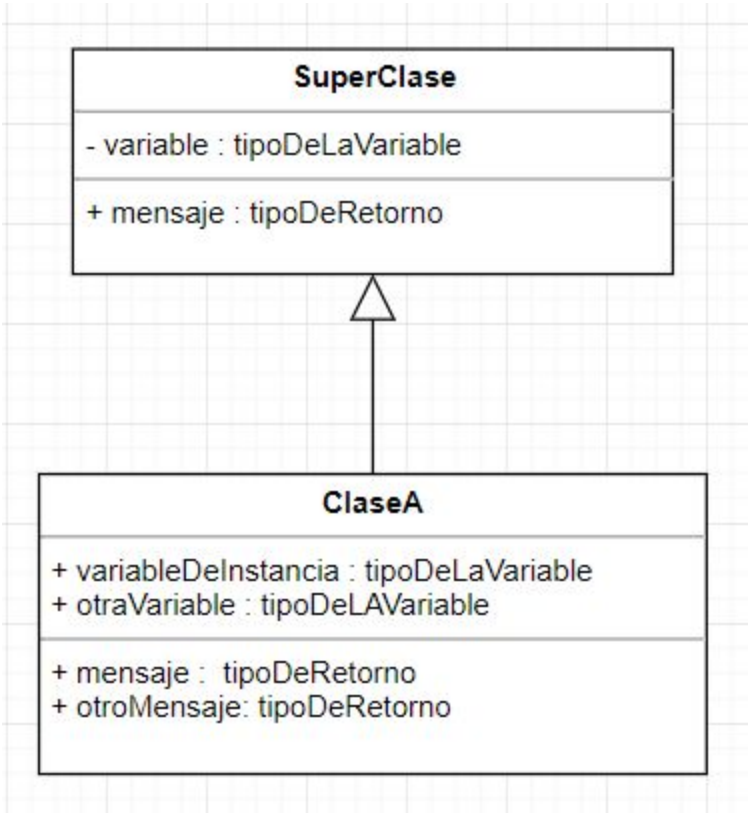
```
public class Alumno {
    private String nombre;
    private LocalDate fechaIngreso;

    public Alumno(String unNombre, LocalDate fecha) {
        this.nombre = unNombre;
        this.fechaIngreso = fecha;
    }

    public String getNombre() {
        return this.nombre;
    }
}
```

Relaciones entre clases

Herencia

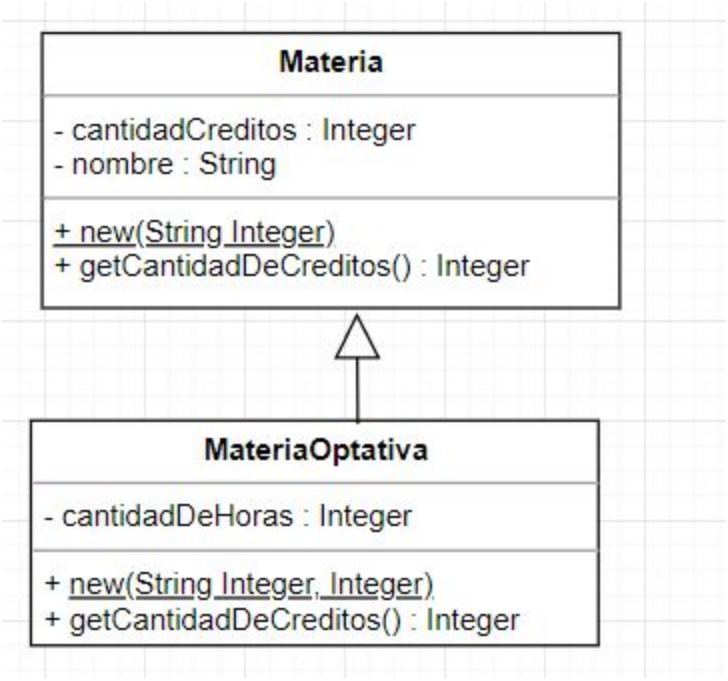
La relación entre una clase y sus subclases se representa en el lenguaje con una flecha. Esta fecha tiene su punta, representada por un triángulo **sin relleno**, señalando a la super clase y su base en la subclase que va a heredar el comportamiento.

Clase abstracta: Identificamos las clases abstractas escribiendo el nombre de la clase en *Italica*



Decimos que la clase **ClaseA** es subclase de la clase **SuperClase**

Modelemos materias dentro de una universidad. De todas las materias conocemos su nombre y la cantidad de créditos que otorga. Pero existen dos tipos de materias, las materias optativas y las materias obligatorias. La materias optativas a diferencia de las obligatorias, tiene una carga horaria y cuando se quiere saber la cantidad de creditos final se multiplica la cantidad de horas por la cantidad de créditos conocidos.



¿Como se ve refleja esto en el código?

Notar que las variables tiene visibilidad protected para que puedan ser vistas por las subclases (es una decisión del programador)

```
public class Materia {

    protected String nombre;
    protected Integer cantidadDeCreditos;

    public Materia(String nombreMateria, Integer creditos) {
        this.nombre = nombreMateria;
        this.cantidadDeCreditos = creditos;
    }

    public Integer getCantidadDeCreditos() {
        return this.cantidadDeCreditos;
    }
}
```

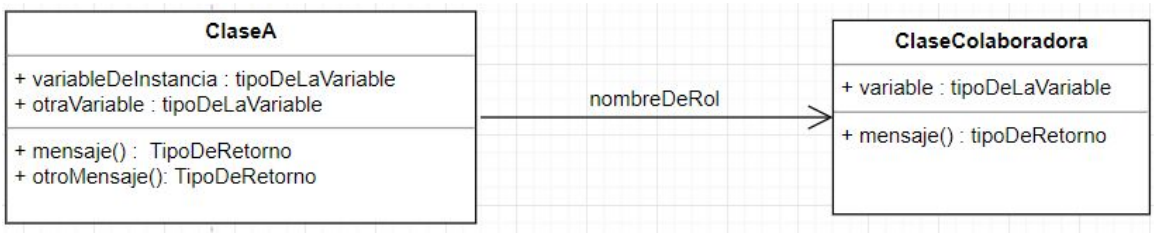
```java
public class MateriaOptativa extends Materia{

    private Integer cantidadDeHoras;

    public MateriaOptativa(String nombreMateria, Integer creditos, Integer horas) {
        super(nombreMateria, creditos);
        this.cantidadDeHoras = horas;
    }

    public Integer getCantidadDeCreditos() {
        return this.cantidadDeCreditos * this.cantidadDeHoras;
    }
}
```

Asociación

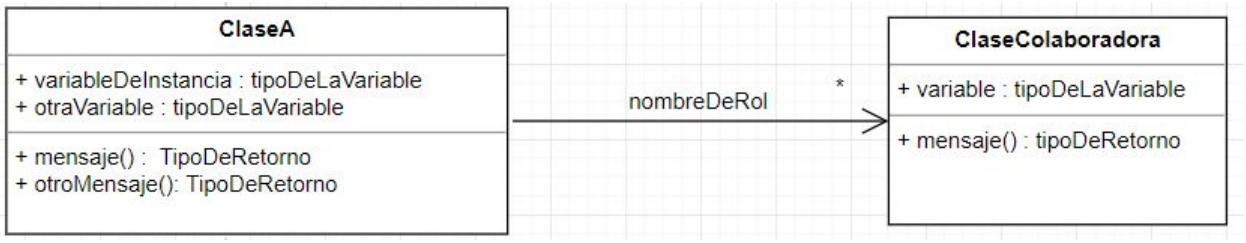
Cuando una clase de nuestro modelo tiene como variable de instancia a otra, la relación entre ellas se representa con una flecha cuya punta representa sin relleno ni base. La flecha apunta a la clase colaboradora y tiene base en la clase que va a recibir la colaboración. Sobre la flecha se escribe el nombre de la variable de instancia (Este nombre se define como el nombre del rol que culple una clase para la otra).

Navegabilidad: Se dice que la **ClaseA** tiene un colaborador interno del tipo **ClaseColaboradora** si en el gráfico la flecha nace de la **ClaseA** y su punta está en la **ClaseColaboradora**.

Cardinalidad: Cuando necesitamos decir que una clase tiene como colaborador interno a una colección de objetos de otra clase, la relación queda expresada con la misma flecha de asociación pero sobre la punta de la fecha le agregamos un asterismo (*).



ClaseA tiene una variable de instancia llamada **nombreDelRol** de tipo **ClaseColaboradora**



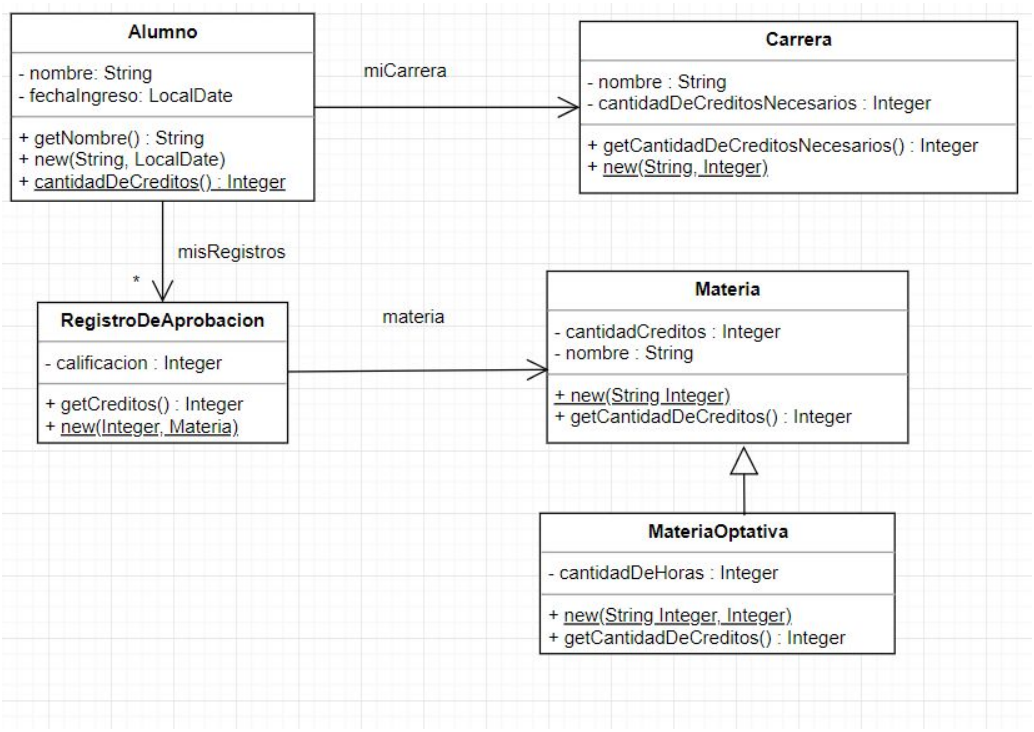
ClaseA tiene como colaborador interno una colección de objetos de **ClaseColaboradora** llamada **nombreDelRol**

¿Como seguimos con nuestro ejemplo?

Los alumnos conocen la carrera en la que están inscriptos. Solo están inscriptos en 1 carrera. De las carreras conocemos el título y la cantidad de créditos necesarios para recibirse.

Además los alumnos guardan una colección de registros de aprobación. De los registros conocemos la calificacion recibida y la materia aprobada. De los registros de aprobación nos interesa conocer la cantidad de creditos recibidos por haber aprobado dicha materia.

Ahora de los alumnos nos interesa conocer la cantidad de créditos que tiene. Veamos nuestro modelo.



Notemos que las variables se llaman *miCarrera* a pesar de que el tipo de la variable es **Carrera** o *misRegistros* a pesar de ser tipo **List<RegistrosDeAprobacion>**

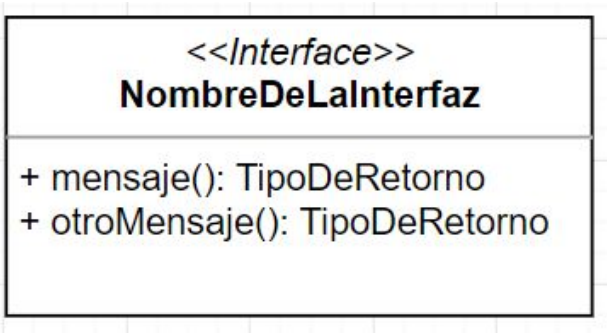
```java
public class Carrera {

    private String nombre;
    private Integer cantidadDeCreditosNecesarios;

    public Carrera(String nombreCarrera, Integer sumarDeCreditos) {
        this.nombre = nombreCarrera;
        this.cantidadDeCreditosNecesarios = sumarDeCreditos;
    }

    public Integer getCantidadDeCreditosNEcesarios() {
        return this.cantidadDeCreditosNecesarios;
    }
}
```

```java
public class RegistroAprobacion {

    private Materia materia;
    private Integer calificacion;

    public RegistroAprobacion(Materia unaMateria, Integer unaCalificacion) {
        this.materia = unaMateria;
        this.calificacion = unaCalificacion;
    }

    public Integer getCreditos() {
        return this.materia.getCantidadDeCreditos();
    }

}
```

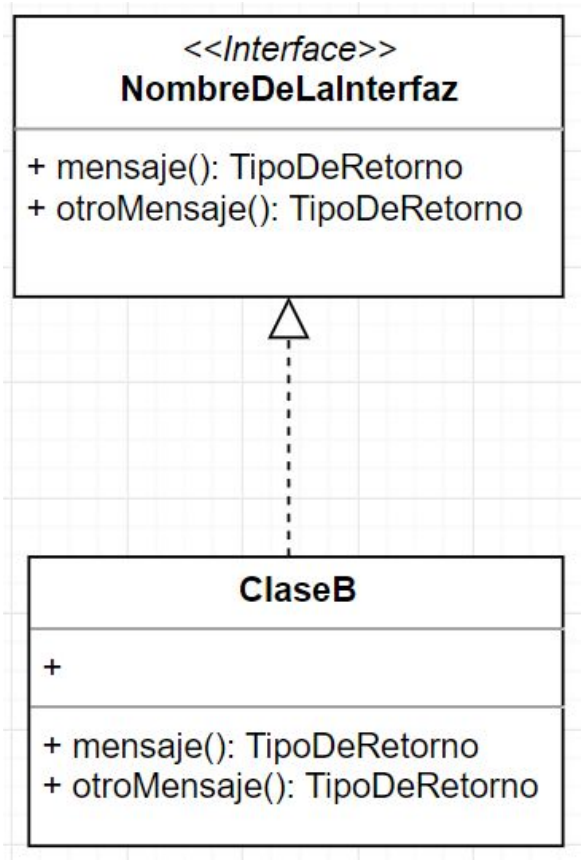
```java
public class Alumno {
    private String nombre;
    private LocalDate fechaIngreso;
    private List<RegistroAprobacion> misRegistros;
    private Carrera miCarrera;

    public Alumno(String unNombre, LocalDate fecha, Carrera unaCarrera ) {
        this.nombre = unNombre;
        this.fechaIngreso = fecha;
        this.misRegistros = new ArrayList<RegistroAprobacion>();
        this.miCarrera = unaCarrera;
    }

    public String getNombre() {
        return this.nombre;
    }

    public Integer cantidadDeCreditos() {
        return this.misRegistros.stream().mapToInt(x -> x.getCreditos()).sum();
    }
}
```

Interfaces

Las interfaces se representan con una caja que solo tiene dos partes. En la primer parte se escribe el nombre de la interfaz y en la segunda parte el nombre de los mensajes que la interfaz define. Algunos editores UML agregan la leyenda <<Interface>> sobre el nombre de la interfaz.



Para indicar que una clase implementa una interface, se debe escribir una flecha punteada que nace en una clase y apunta a una interfaz



La **ClaseB** implementa la interfaz **NombreDeLaInterface**

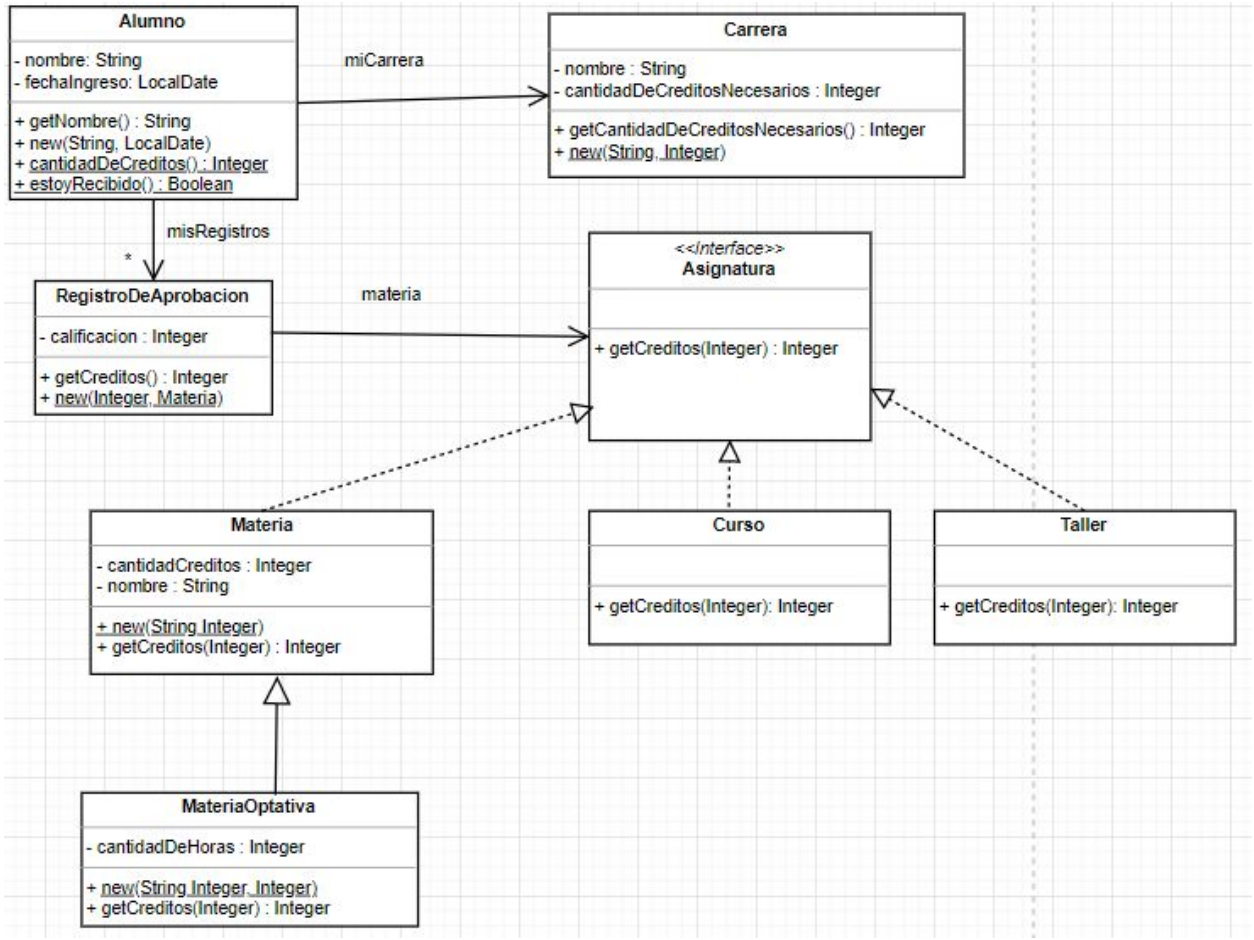
¿Y que pasa con las universidades?

Modelemos un poco más. Agreguemos cursos y talleres. Los cursos y talleres son como las materias ya que también otorgan creditos. Los cursos multiplican por dos la cantidad de creditos que la calificación que se saque el alumno. Por ejemplo: Si apruebo un curso con 10, sumare 20 creditos. Los talleres solo suman 1 crédito.

Definamos la interfaz Asignatura que define el mensaje *getCredito* que recibe como parámetro una calificación.

Además agreguemos la responsabilidad al alumno de saber si esta recibido o no. Comparando la cantidad de créditos obtenido con los requeridos por la carrera.

¿Como queda nuestro modelo? Adaptemos nuestro modelo a los nuevos requerimientos.



Notar que el nombre de la variable *materia* no cambio a pesar de que ahora el tipo de la variables es **Asignatura**

```java
public interface Asignatura {

    public Integer getCredito(Integer unaCalificacion);

}
```

```java
public class Taller implements Asignatura{

    @Override
    public Integer getCredito(Integer unaCalificacion) {
        return 1;
    }

}
```

```java
public class Curso implements Asignatura{

    @Override
    public Integer getCredito(Integer unaCalificacion) {
        return unaCalificacion * 2;
    }

}
```

```java
public class Materia implements Asignatura{

    protected String nombre;
    protected Integer cantidadDeCreditos;

    public Materia(String nombreMateria, Integer creditos) {
        this.nombre = nombreMateria;
        this.cantidadDeCreditos = creditos;
    }

    @Override
    public Integer getCredito(Integer unaCalificacion) {
        return this.cantidadDeCreditos;
    }
}
```

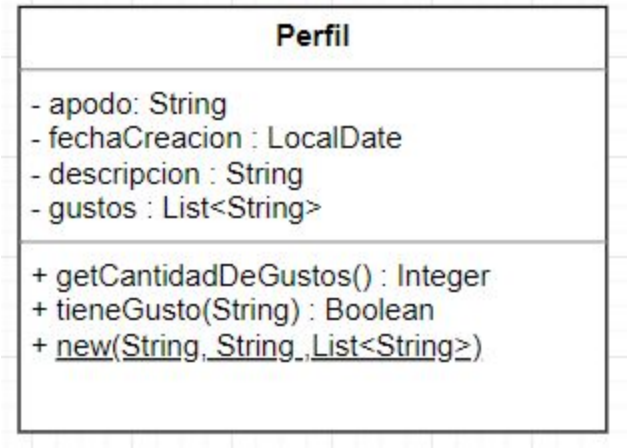
```java
public class MateriaOptativa extends Materia{

    private Integer cantidadDeHoras;

    public MateriaOptativa(String nombreMateria, Integer creditos, Integer horas) {
        super(nombreMateria, creditos);
        this.cantidadDeHoras = horas;
    }

    @Override
    public Integer getCredito(Integer unaCalificacion) {
        return this.cantidadDeCreditos * this.cantidadDeHoras;
    }
}
```

```java
public class Alumno {
    private String nombre;
    private LocalDate fechaIngreso;
    private List<RegistroAprobacion> misRegistros;
    private Carrera miCarrera;

    public Alumno(String unNombre, LocalDate fecha, Carrera unaCarrera ) {
        this.nombre = unNombre;
        this.fechaIngreso = fecha;
        this.misRegistros = new ArrayList<RegistroAprobacion>();
        this.miCarrera = unaCarrera;
    }

    public String getNombre() {
        return this.nombre;
    }

    public Integer cantidadDeCreditos() {
        return this.misRegistros.stream().mapToInt(x -> x.getCreditos()).sum();
    }

    public Boolean estoyRecibido() {
        return this.miCarrera.getCantidadDeCreditosNecesarios() < this.cantidadDeCreditos();
    }
}
```

Más Ejemplos!

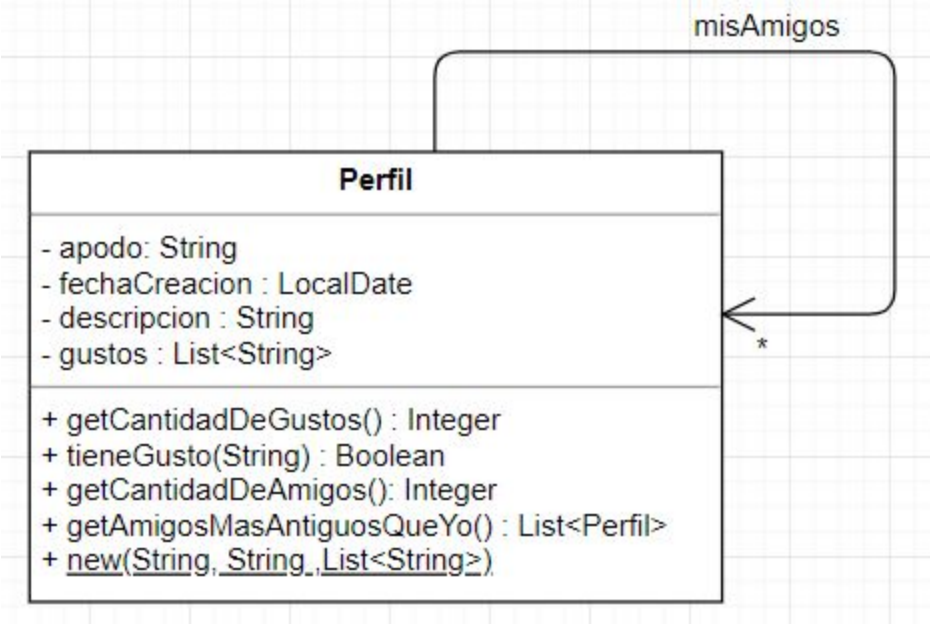
Perfiles

Estamos modelando una clase Perfil para una nueva red social. A los perfiles los identificamos con un apodo. Además cada perfil guarda la fecha de creación del perfil, una descripción y una colección de strings que identifican sus gustos. De los perfiles nos interesa saber la cantidad de gustos y si ese perfil tiene un gusto particular.



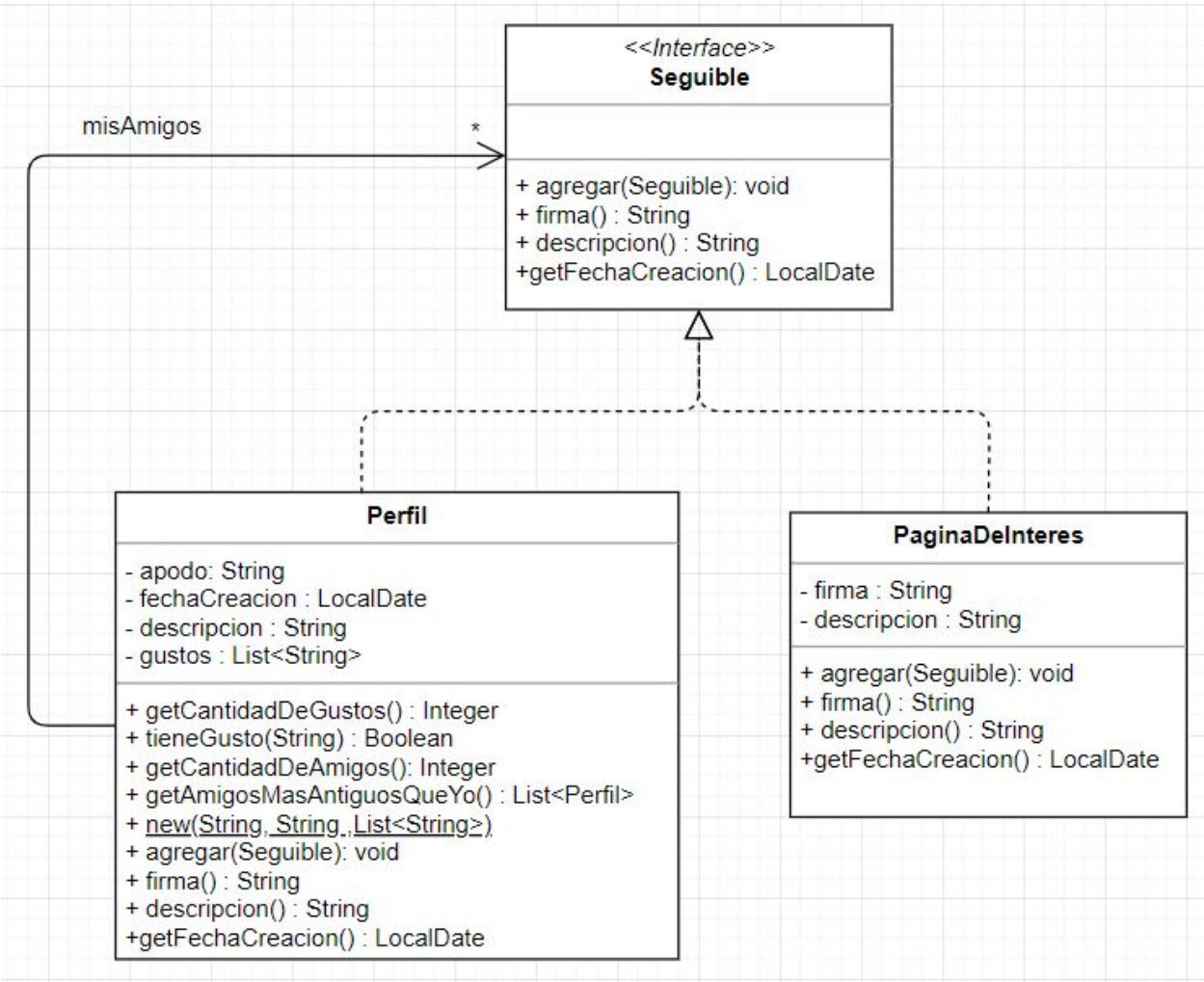
Perfiles y amigos

Continuamos con los perfiles y ahora se nos pide que los perfiles tengan una coleccion de otros perfiles amigos. Nos interesa conocer la cantidad de perfiles amigos que tiene un perfil y los perfiles amigos que hayan sido creados antes que el perfil actual.



Perfiles y seguidores

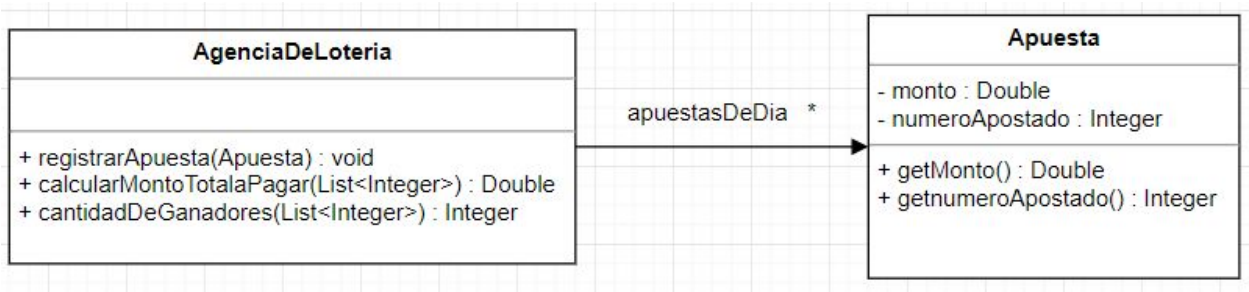
Ahora nuestros perfiles pueden tener como amigos paginas de interes ademas de otros perfiles. Para ello, definimos la interfaz Seguible que define el comportamiento de poder agregar Seguible, devolver una firma, una descripción y la fecha de creación. Las páginas de interés y los perfiles implementan la interfaz Seguible . Los perfiles response su apodo cuando le mandan el mensaje *firma()* mientras que las paginas de interes responden la fecha de hoy cuando le envían el mensaje *getFechaCreacion()* y no realizan ninguna acción cuando reciben el mensaje *agregar*



Pagar las apuestas

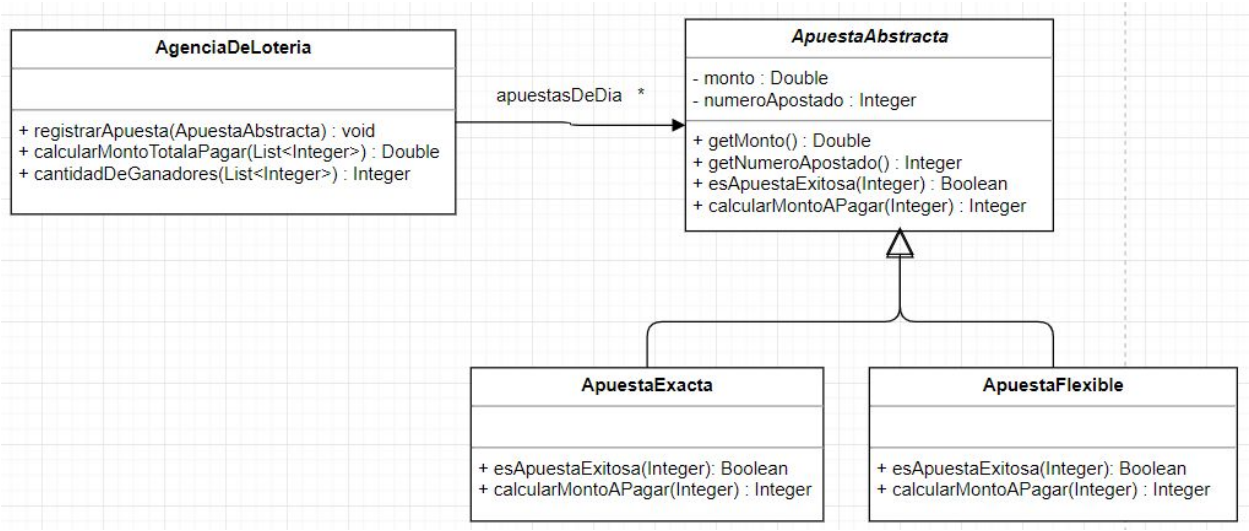
Una agencia de loteria recibe apuestas durante todo el dia. Las apuestas están conformadas por por un número entre 0 y 99 y un monto, que es la plata que las personas apuestan. Al finalizar el dia la agencia recibe una lista de números que son los números que salieron sorteados. Se nos contrata para realizar un programa que nos permita registrar apuestas y una vez recibida la lista de números ganadores poder:

- Identificar cuantas apuestas ganadoras hay.
- Cuanto es el monto a pagar. Para calcular el monto a pagar, se debe multiplicar el monto apostado por 70.



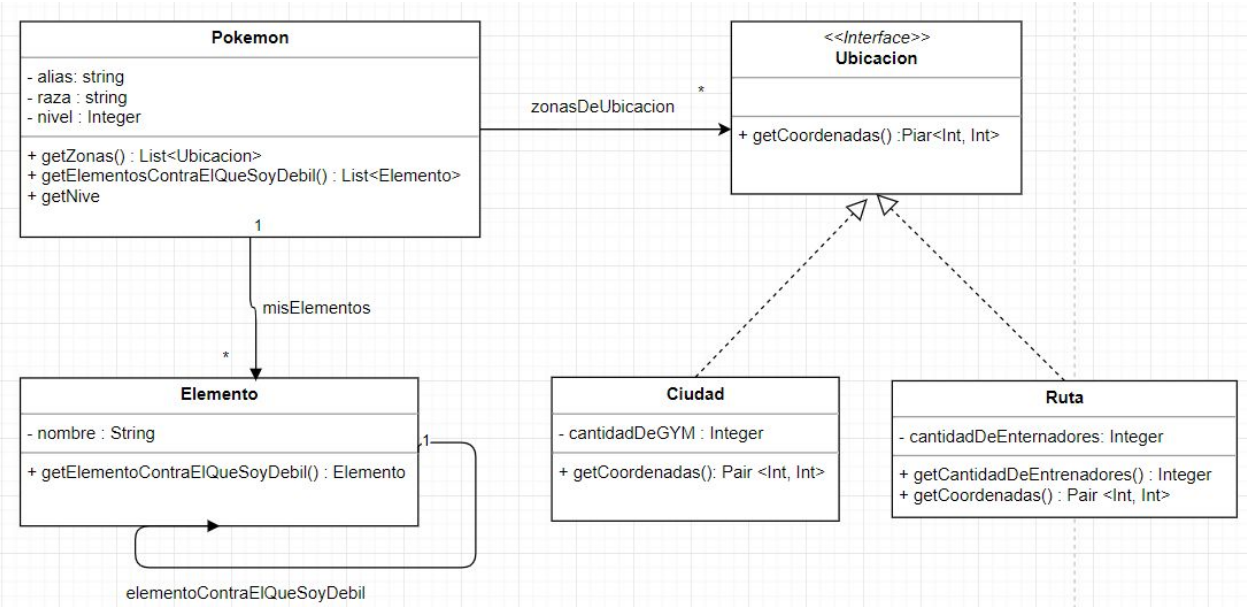
Pagar las apuestas 2

Nuestra aplicación anterior fue un éxito y la agencia de loteria nos propone agregar un nuevo tipo de puesta. Esta apuesta es más flexible que la anterior, puesto que para considerarla ganadora es el número ganador puede ser el número exactamente apostado o 3 números menos o más. Por ejemplo: Si apuesto el número 25 y sale el número 27, mi apuesta será considerada ganadora de todos modos. Modificamos el modelo anterior para cumplir con los nuevos requerimientos. Lo malo de esta puesta es que al momento de calcular el monto a pagarse debe calcular el monto apostado por 50.



Atrapalos ya!

Conocemos la clase Pokemon. Los Pokemón tienen un alias del tipo String, una raza del tipo string, un nivel de tipo Integer y una colección de elementos. Cada Elemento conoce su nombre de tipo string y el elemento contra el que es débil. Por otro lado, cada Pokemón conoce una lista de zonas donde podemos encontrarlos para capturarlos. Estas zonas son una colección de objetos que implementen la interfaz Ubicable. La interfaz Ubicable define el mensaje *getCoordinadas* el cual retorna un par de enteros. Las Ciudades y las Rutas implementan la interfaz Ubicable, de las ciudades conocemos la cantidad de gimnasios pokemon que hay en ella y de las rutas conocemos la cantidad de entrenadores que están disponibles para atacar.



Y el código?

Podes codificar todos los modelos planteados en los ejemplos y si tenes dudas podes consultarlas con la cátedra durante las prácticas, las listas o el grupo de Telegram.

Pero ese modelo no me parece el más acertado.

Podes plantear un UML con tu modelo de solución y discutirla con la catedra