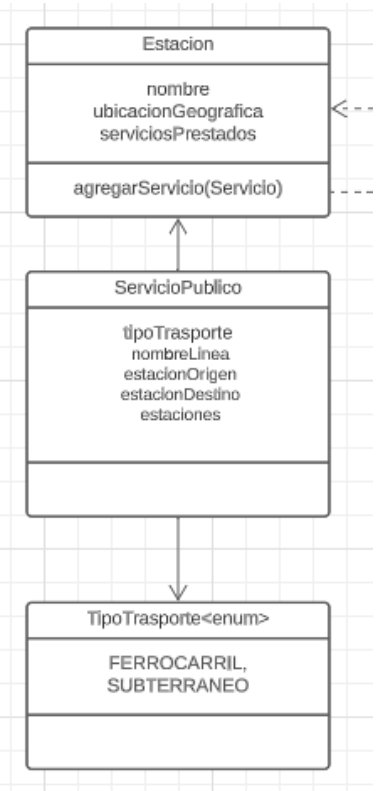
Decisiones de diseño tomadas:



Servicios públicos

En primer lugar, creamos una clase Servicios Públicos que conociera un tipoTransporte y un nombreLinea

El **TipoTransporte** decidimos representarlo con un ENUM. *Justificación*: *Al ser elementos bien definidos, reducidos y por el momento sin comportamiento, consideramos que el tipo enum es el indicado. Por el momento el ferrocarril y subterráneo son los únicos servicios públicos definidos dentro de nuestro dominio, por lo que dentro de* ***ServicioPublico*** *es importante diferenciarlos y saber de qué tipo es el servicio público del cual se está hablando*.

Para representar la línea, teníamos dos opciones:

1) Que la clase Servicios Públicos conociera un **tipoLinea** como atributo, y a su vez este tipo línea contendrá los atributos su nombre, su estación de origen, estación de destino y el conjunto de estaciones que la conforman, separando una línea de un servicio público

2) Que la clase Servicios Públicos conociera directamente los atributos nombre, su estación de origen, estación de destino y estaciones que la conforman, junto con el tipo de transporte.

Al final, para representar al servicio público decidimos poner todos los datos de la línea en una sola clase **ServicioPublico** incluyendo al **tipoTransporte** y al **nombreLinea**. *Justificación: decidimos priorizar la simplicidad dado que una línea forma parte tanto de un ferrocarril como de un subterráneo según sea el caso, por ende el poner sus atributos dentro de un* ***ServicioPublico*** *nos pareció lo mismo que ponerlos dentro de otra clase llamada quizá Línea.*

La lista de **estaciones** contendrá todas las estaciones por las que pasa el servicio en orden de pasada, siendo la primera **estacionOrigen** y la última **estacionDestino**. *Justificación: en el dominio esto está bien diferenciado.*

Si sabemos que los servicios públicos se darán de altas solamente por el administrador, entonces debemos confiar que los datos ingresados por el mismo serán válidos. Por eso decir que la lista de estaciones incluyen al origen y destino sin necesidad de hacer una validación.

Cada estación tendrá su propia clase en la que se definirá su **nombre, ubicacionGeografica, serviciosPrestados** y el método **agregarServicio().** Cada una, es conocida por el **ServicioPublico** al que pertenece. *Justificación: cada* ***ServicioPublico*** *hace referencia a un*

tipoTransporte *y éste indistintamente del tipo, posee una estación que debe conocer al momento de crearse el servicio.*

Dado que una estación puede pertenecer a más de una línea, lo recomendable sería que las estaciones estén previamente cargadas al sistema al momento de construir un servicio público.

De momento, cada vez que se instancie un servicio público, este tendrá un recorrido solamente de ida a través de sus estaciones. En caso de ser un viaje de ida y vuelta, debería representarse con dos Servicios públicos distintos, en donde la estación inicial de uno será el final del otro y viceversa. De momento ese comportamiento no se definió dado que no se cuenta con la información suficiente.

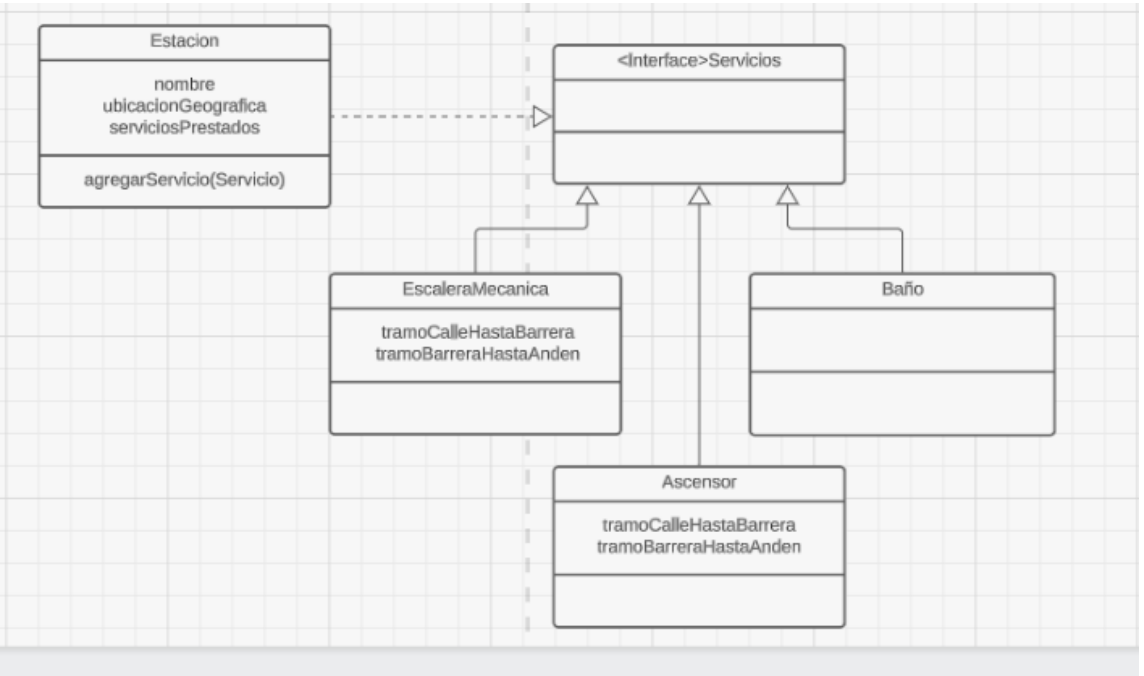
Servicios

Por otra parte los servicios prestados por cada estación son representados en una interfaz "**Servicios**". Hasta ahora tenemos tres tipos de servicios los cuales son representados por las clases: "**EscaleraMecanica**", "**Baño**" y "**Ascensor**".

Tanto **EscaleraMecánica** como **Ascensor** poseen los atributos **tramoCalleHastaBarrera/ tramoBarreraHastaAnden.**

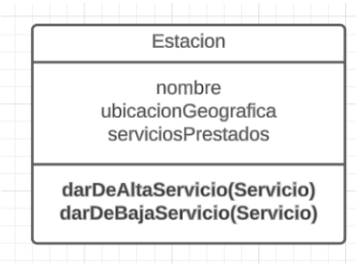
*Justificación: Si bien por el momento estos servicios no poseen comportamiento, consideramos que crear **Servicios** como interfaz, nos sería útil en un futuro para poder continuar agregando servicios en caso de que los haya. De igual manera, el implementarlos como interfaz nos da la posibilidad de que aunque cada uno se comporte de manera diferente entre si, tambien podrian repetir comportamiento y atributos, sin necesidad de repetir código*

Por el momento no conocemos cuál será la aplicación de estos "tramos", pero sí sabemos que serán parte de ambos medios de elevación, por lo que temporalmente serán tipo String.



Prestación de servicio

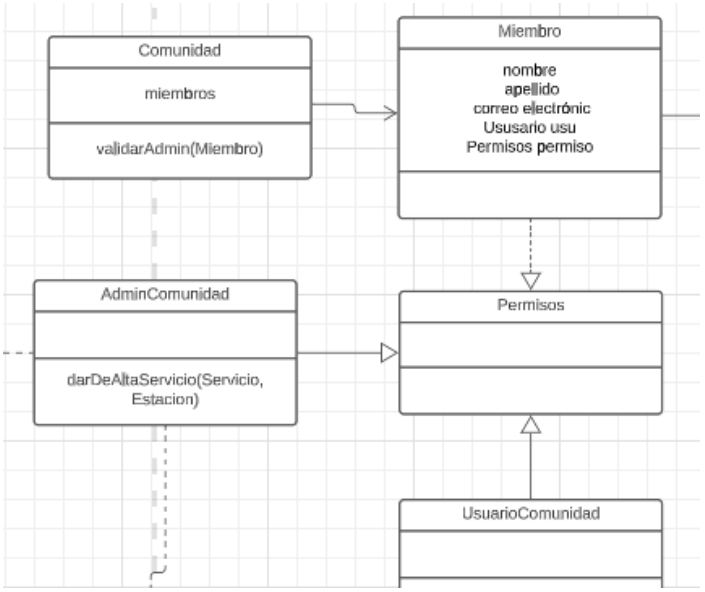
Una estación brindará cero, uno o más servicios dentro de su dominio, representada con una colección **serviciosPrestados** que se guardaran como atributo.
Es por ello que esta interfaz estará conectada a cada estación Estación, quien será la que podrá implementarla.



Justificación: Es una estación quien ofrece o no un servicio. ***(serviciosPrestados)***

Para dar de alta y baja servicios se agregaran los métodos **darDeBajaServicio** y **darDeAltaServicio**, que removerán o agregaran elementos de la lista de serviciosPrestados según corresponda.

Comunidades y miembros

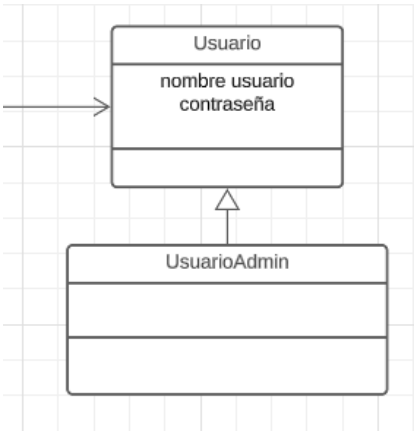


Representamos una **Comunidad** como una clase**,** que conoce una colección de miembros que la conforman, y como método, una comunidad puede **validarAdmin(),** de su comunidad.
Justificación: una comunidad no sería una comunidad sin sus miembros. Dado que por el momento no sabemos quien es quien validará el administrador de la misma, asumimos que será la misma comunidad quien decida quién será su administrador.

Por otro lado diseñamos a cada **Miembro** como una clase conocida por **Comunidad** por los motivos antes mencionados. Esta clase tiene como atributos **nombre, apellido, correo electrónico, usuario y permiso.**

Justificación: todos sus atributos caracterizan al objeto como tal, pero queremos hacer hincapié en el atributo ***permiso****. Pensamos que un miembro de por sí es un "usuario común", con la diferencia (además de pertenecer a una comunidad) de que este puede en algún momento pasar a ser administrador. Por lo que para tener una manera de verificar esta distinción, decidimos agregarle un "****permiso****" a cada miembro. Según sea el permiso, este calificará como* ***AdminComunidad*** *o* ***UsuarioComunidad.***
AdminComunidad tiene el método **darDeAltaServicio()**, *Justificación: Tal como figuraba en la descripción del dominio, serán las comunidades quienes den de alta nuevos servicios en caso de requerir, en este caso consideramos que será el administrador de la misma.*

Usuario de la plataforma:



Finalmente tenemos la clase **Usuario** que tiene como atributos **nombre usuario y contraseña** y es conocida por la clase **Miembro**.
Justificación: un miembro es un usuario

Modelamos al **UsuarioAdmin** como una clase que representará al administrador de la plataforma y hereda de **Usuario.**
*Justificación: el **usuario admin** tendrá todos los atributos y comportamiento de un **usuario** y más.*

Seguridad:

En primer lugar, necesitamos dentro de una clase los siguientes mensajes:

1) Iniciar sesión
2) Registrarse

Por lo que decidimos crear una clase Validador() guarde los siguientes atributos:
-password
-username
A su vez, empezamos a encarar los siguientes mensajes:

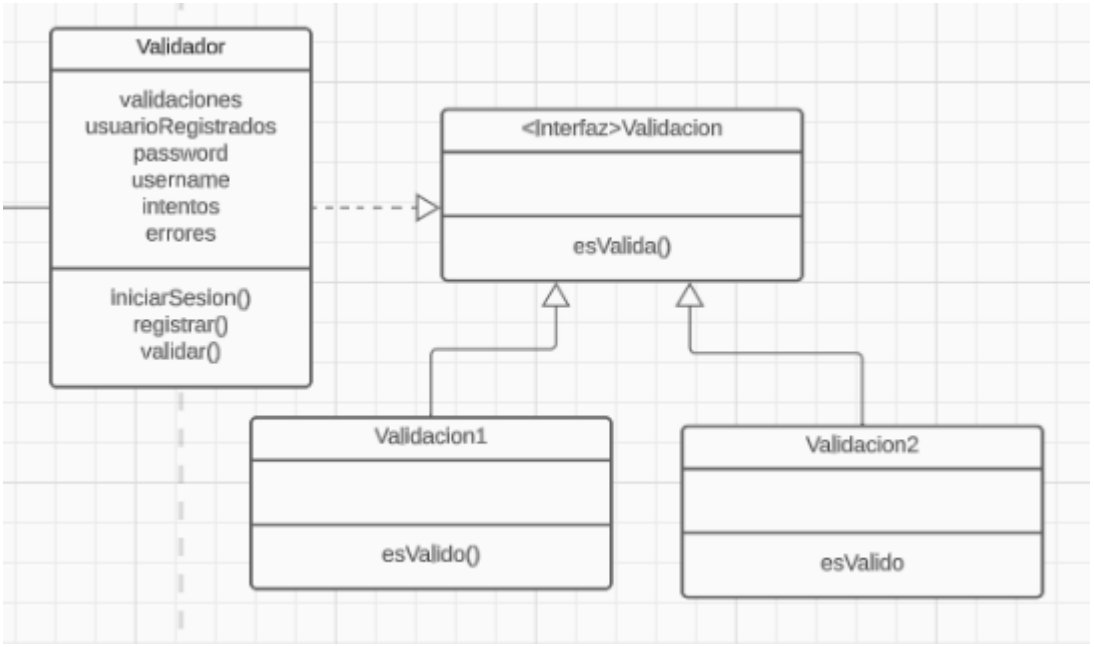
1) Para iniciar sesión, necesitamos que se cumpla los siguientes requisitos:

- *Que el usuario ya se haya registrado anteriormente:* para ello cada vez que el usuario se registre, este debe guardarse dentro de una colección mutable **usuariosRegistrados** desde donde se podrá filtrar al usuario cuyo username coincida. Para ello es importante que la clase Usuario tenga getter
- *Que no haya superado la cantidad de intentos fallidos:* para ello se deberá agregar un atributo **intentos** (por defecto la inicializamos con el valor 3), que guarde la cantidad de intentos que tenemos para iniciar la sesión. Para cada intento fallido se le restará una unidad.

  Como no sabemos qué comportamiento produce un inicio de sesión, decidimos que de momento devuelva una clase usuario, y en caso de no encontrar el usuario, devolverá una excepción no chequeada

2) Para registrarse, primero chequea que el password que tenemos como atributo cumpla con todas las validaciones dados de alta en el sistemas. Estas validaciones se guardarán en una colección de **validaciones.** Los elementos del mismo será de una interfaz “Validacion” cuyo metodo **esValido()** sera implementada por sus clases hijas

   Justificación: Consideramos realizar un patron strategy dado que todas las validaciones implementaran un mismo mensaje esValido (booleano), pero a su vez, cada clase hijo redefinirá el método con un comportamiento propio de cada subclase



A la hora de registrarse, se delega a un mensaje **validar()**, tipo booleano, que para cada elemento de la colección de validación le preguntará por dicho password. De no cumplir dicho requerimiento, devolverá una excepción no chequeado distinta para cada validación que falle que se atraparan y guardaran en una lista de **errores**, la cual se mostrará en pantalla una vez que terminen los chequeos de password. Por otro lado si supera las validaciones, se creará un nuevo usuario el cual se guardará en la lista usuariosRegistrados. Este usuario tendrá, de momento, atributos inmutables.