# 

# **Decisiones de Diseño**

## Estaciones y Servicios

Al diseñar nuestro código, tomamos algunas decisiones de diseño importantes. Para la clase "Línea", decidimos que cada línea tendría un ArrayList de estaciones y un enum que indicaría el tipo de transporte (subterráneo o ferrocarril). El enum se decidió debido a que suponemos que la línea no tendrá un comportamiento basado en su tipo. Además, hicimos los métodos de la estación de origen y destino ya que la lista de estaciones está ordenada, de ahí el uso de una ArrayList y sería redundante almacenar estas estaciones.

Para la clase "Estación", decidimos que tendría un nombre, una ubicación geográfica (que consiste en una lista con 2 floats, latitud y longitud) y, para lograr esto, creamos la clase "Coordenadas". Además, cada estación tendría un set de servicios. Decidimos usar un set ya que el servicio no debería poder repetirse y el orden no importa.

Finalmente, decidimos crear una clase abstracta llamada "Servicio". En lugar de una interfaz, elegimos una clase abstracta ya que las clases que la heredan tienen el mismo método no abstracto que funciona exactamente igual. Los servicios "Baño", "Escalera Mecánica" y "Ascensor" heredan de esta clase. La clase "Servicio" tiene un atributo "funciona" que indica si el servicio en sí mismo funciona.

## Comunidades y miembros

Al momento de crear las comunidades primero se tomó en cuenta que las personas que las integran, dado que las comunidades en sí son un conjunto de personas.

Para las personas se creó una clase teniendo en cuenta los datos que son pedidos para cada uno de ellos, siendo estos el nombre, apellido, correo electrónico y crear un método para instanciar estos datos del mismo.

para las comunidades se instaló como variable un arraylsit de personas para contener a todas las personas que lo integran, junto a otro método que agrega a las personas a la lista de integrantes de la comunidad.

## Usuarios

Pensamos que la clase usuario debe tener la responsabilidad de almacenar los datos del usuario, como el nombre y la contraseña, y no debe ser responsable de validar la complejidad de la contraseña.

Si se realiza la validación dentro de la clase usuario, estaríamos mezclando dos responsabilidades diferentes en la misma clase, lo que dificultará el mantenimiento y la evolución del sistema. En cambio, utilizando una clase de validador de contraseñas, se logra una separación clara de responsabilidades.

También, al crear una contraseña, terminamos decidiendo que en el setter de la contraseña se utilice al validador de contraseña de la clase mencionada previamente. De esta manera, se notifica al usuario que ha ingresado una contraseña inválida a través de una excepción y se evita que se guarde una contraseña no segura en el objeto Usuario. Decidimos realizar la validación únicamente en el setter y no en el constructor ya que puede suceder que al crear un usuario aún no se tenga disponible la contraseña para validar.

Por último, tener la validación en el setter nos permite su modificación y asegurarnos de que el Usuario siempre tenga un estado válido

## Contraseña

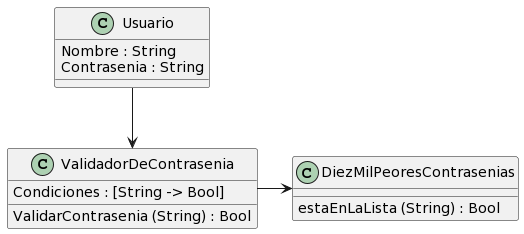
La idea desde el principio era usar una lista de funciones para modelar las condiciones que se necesitan a la contraseña, ya que esta forma es adaptable y sería fácil agregar nuevas condiciones. Después se usaría una operación all para verificar que la contraseña cumpla todas las condiciones. Esto se agrupó en una clase cuya función principal era, a través de un método, tomar una contraseña y devolver un booleano que determina si es válida.

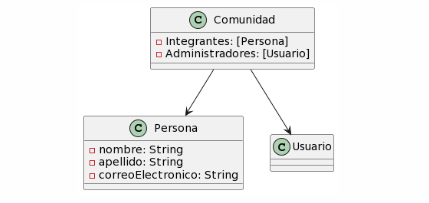
Fuera de esto, la complicación era la lista de 10 mil contraseñas, ya que había que sacarla de un documento externo. Como esto tenía cierto grado de complejidad adicional, se decidió separarlo en otra clase que se ocupe de todo lo relacionado con esta lista.

El primer intento implicaba pegar toda la lista como un array directamente en el código. La misma debía estar ordenada para la eficiencia de la búsqueda. El problema con esto fue que el archivo java era demasiado pesado para correr, por lo que esta alternativa fue descartada.

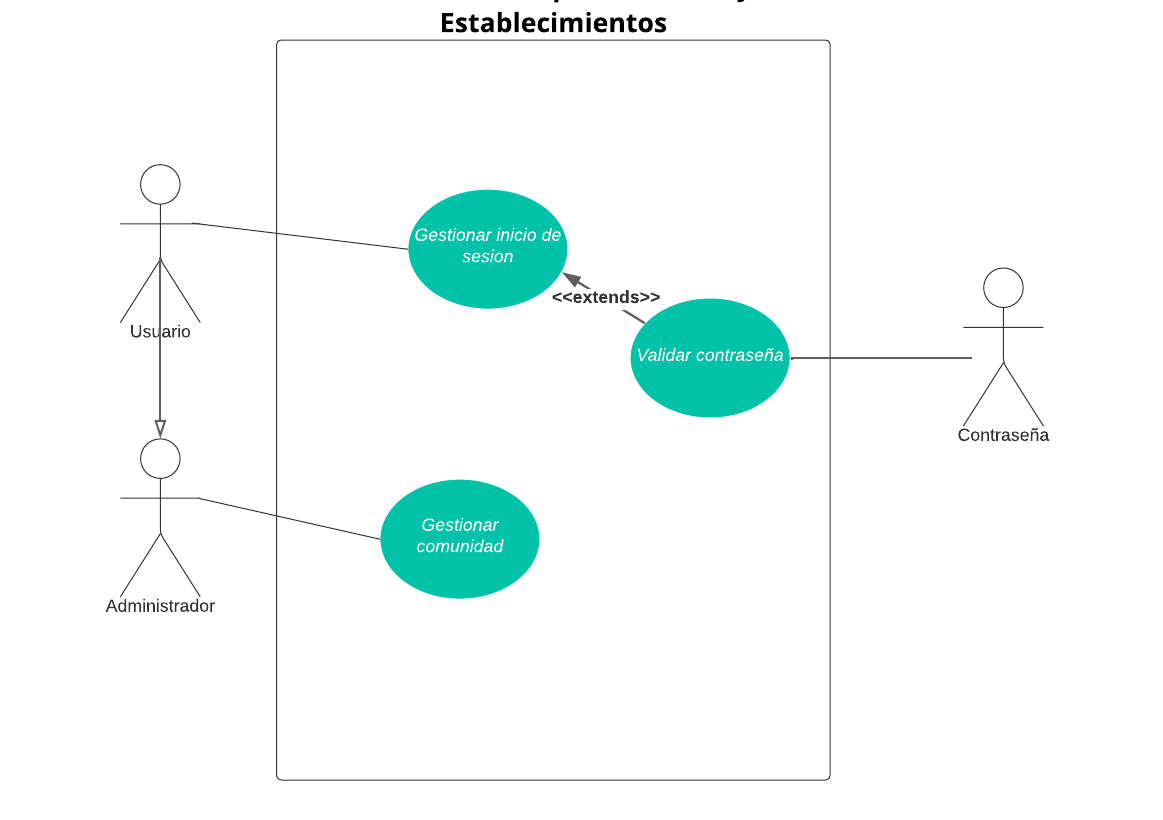
Lo que quedaba era leer directamente del archivo. Como el archivo ya estaba ordenado, se pudo aplicar una función que deja de buscar cuando se pasa la posición en la que la string tendría que haber estado.

# **Diagramas**

****

****

# **Caso de Uso**

****