UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Buenos Aires

Diseño de Sistemas

–2023–

Docentes: Ezequiel Escobar, Gaston Prieto

Tutor: Lucas Saclier

Trabajo Práctico Anual: Primera Entrega

Monitoreo de Estado de Servicios de Transporte Público y de Establecimientos

Localización: Medrano		**Curso: K3001/K3101**	
Legajo	**Apellido y Nombre**		
203.348-3	**Alberio,** Valentina		
203.680-0	**Gobbi,** Micaela Nicole		
203.761-0	**Larocca**, Lourdes Florencia		
204.620-9	**Sandoval,** Aylén Martina		
204.091-8	**Soteras,** Martina Rocio		
Entrega/Revisión	**1**	**2**	**3**
Fecha Entrega	**25/04/2023**		
Firma Docente			

Requerimientos generales

1. Se debe permitir la administración de servicios públicos (en adelante se llama "*administración*" a las acciones de alta, baja y modificación).
2. Se debe permitir la administración de servicios.
3. Se debe permitir la administración de prestación de servicios.
4. Se debe permitir la administración de comunidades y miembros.

Administración de Servicios Públicos:

En este momento del diseño, se entiende a los servicios públicos como bienes y actividades cuyo objetivo es garantizar la movilidad de los ciudadanos. Por esta razón, debe contener una línea de transporte a la que pertenece y un tipo de transporte (subterráneo o ferrocarril). Se tomó la decisión de que *LíneaDeTransporte* sea una clase dado que la primera cuenta con diferentes atributos relacionados con las estaciones (que en nuestra solución, refieren a una instancia de la clase Estación) y que *TipoDeTransporte* sea una interfaz (y que actualmente las clases Subterráneo y Ferrocarril la implementen), para garantizar la extensibilidad del modelo.

Para permitir que se realicen las acciones de alta, baja y modificación de un Servicio Público, se ha considerado que en una comunidad, los Administradores son los encargados de dar de alta o de baja un Servicio Público, mientras que la modificación implica que esté habilitado o deshabilitado. Esto último se refiere a cuando, en la vida real, un servicio de transporte está suspendido por obras. Cabe aclarar que es limitación de esta solución modificar la línea de transporte, puesto que en la vida real podría cambiar de concesión un servicio público y modificar su denominación.

Administración de Servicios

Se interpreta por servicios a aquellas prestaciones que se presentan a modo de satisfacer necesidades de los consumidores de Servicios Públicos. Estos servicios pueden ser definidos por una comunidad dentro de la plataforma, o estar presentes en forma estándar en la misma. Para modelarlos correctamente, se decidió crear la clase *Servicio* con los atributos *tipoDeServicio*, *descripción* y *estaHabilitado*. En un principio, los dos primeros atributos mencionados corresponden al tipo de dato String, ya que con la información que disponemos actualmente no se puede identificar responsabilidades en los mismos.

Con la descripción de un servicio se busca contemplar que en los servicios de medios de elevación (escalera o ascensor) se tome en cuenta los tramos descritos en el enunciado.

La alta y baja de los servicios, entonces, pueden ser realizadas por las comunidades con los métodos correspondientes. Por el momento, se interpreta a la modificación como a la posibilidad de que las comunidades cambien las descripciones de los servicios. Para este objetivo, se desarrolló el método *modificarDescripcion* dentro de la comunidad.

Administración de prestación de Servicios

Se considera que los Administradores son los encargados de habilitar y deshabilitar los servicios, con el propósito de informar cuáles de ellos están siendo prestados y cuáles se encuentran inhabilitados para su uso.

En este caso, la modificación de un servicio es cambiar la descripción del mismo, de lo cual se encargarán los Administradores como se explicó anteriormente. Por otro lado, a la hora de evaluar la prestación del éste se querrá saber si el mismo se encuentra disponible para ser utilizado o no y es lo que concierne a la modificación.

Administración de Comunidades y Miembros

Para resolver las funcionalidades de las comunidades y los miembros, modelamos la clase *Usuario* que está asociada con las clases que implementan la interfaz *Rol.* De momento, existen dos: *Administrador*, que nuclea la funcionalidad que le corresponde a este tipo de usuario, y *Miembro*, que aún no tiene funcionalidad definida. Se creó una lista de roles, debido a que un *Usuario* podría tener más de un rol si es que se encuentra en distintas comunidades. Además, las responsabilidades que adquiere un Usuario al unirse a una Comunidad deben ser delegadas en otra clase para poder asegurar la cohesión del modelo.

Se ha decidido modelar a las comunidades en la clase *Comunidad*, que tienen como atributo una lista de instancias de la clase *Servicios*, representando los servicios que la comunidad considera relevantes y necesita información sobre su estado. *Comunidad* además tiene como atributo una lista de *Usuario*, que representa a todos los usuarios que pertenecen a dicha comunidad; independientemente de los roles que cumplen.

Las comunidades se encargan de administrar la prestación de servicios y actualizar la información disponible. Estas tareas se encuentran delegadas en los Administradores.

Requerimientos de seguridad

1. El sistema debe permitir el registro de usuarios.
2. Validar la no utilización de credenciales por defecto de software.
3. Verificar que las contraseñas de los usuarios no se encuentren entre las 10.000 contraseñas más comunes.

4. Clasificar la seguridad de las contraseñas en fuerte, moderada y débil, notificando al usuario.
5. Rechazar las contraseñas que no cumplan los requisitos de longitud y variedad de caracteres, notificando al usuario.

La responsabilidad que implica permitir el registro de usuarios guardando, por el momento, un usuario y contraseña es delegada a la clase *Registro.* Su método principal es *registrarUsuario,* en el cual se crea una instancia de la clase Usuario pasando por parámetro un nombre de usuario, una potencial contraseña y un correo electrónico, dejando los demás atributos a ser modificados por el propio usuario. Previo a esto, la comprobación de que la contraseña ingresada sea válida (cuyas condiciones se detallan más abajo) y que no haya otro usuario registrado con el mismo nombre, implica añadir a este usuario creado a la lista de usuariosRegistrados. Esta lógica es momentánea, debido a que al persistir en un futuro los usuarios, no haría falta dicha colección.

La clase *Contraseña* tiene de atributos la contraseña en sí, modelada con un String, el usuario que la ha utilizado y una lista denominada validador, donde se encuentran todas las condiciones que requiere una contraseña para ser válida.

En nuestro modelo, existe una clase por cada característica que debe cumplir la contraseña (longitud entre 8 y 64 caracteres, no contener repeticiones de un mismo carácter, no utilizar credenciales por defecto ni utilizar una contraseña demasiado común), a las cuales se les delega la verificación de estos criterios. Todas ellas implementan una interfaz denominada Condición.

Estas clases no solo permiten que se sepa si una contraseña es válida cuando cumple todas estas condiciones o si hace falta reingresarla, sino que a su vez lanzan excepciones cuando su característica no se cumple. Esto le permite al usuario saber qué debe modificar en su contraseña. Se ha decidido extrapolar las validaciones en las clases para mejorar la extensibilidad del código: fácilmente se puede añadir o remover un criterio de validez.

A su vez, para informar al usuario qué tan fuerte es su contraseña, se decidió utilizar el patrón State mediante la creación de las clases *Débil*, *Moderada* y *Fuerte*, las cuales heredan de la clase abstracta *MedidorDeFuerza.* La misma fue diseñada de esta manera puesto que el método fuerza tiene comportamiento, lo cual consiste en mostrar la fuerza que tiene la contraseña que el usuario ha ingresado para registrarse por lo que no podría ser una interfaz. La forma de tratar los criterios de transición de estados fue desarrollada de manera modular con el objetivo de facilitar futuras modificaciones, en caso de que resulte necesario. Dentro de cada clase perteneciente al patrón están especificados los métodos con las condiciones que deben cumplirse para transicionar de estado.