UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Buenos Aires

Diseño de Sistemas

–2023–

Docentes: Ezequiel Escobar, Gastón Prieto

Tutor: Lucas Saclier

Trabajo Práctico Anual: Tercera Entrega

Monitoreo de Estado de Servicios de Transporte Público y de Establecimientos

Sede: 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**	**06/06/2023**	**11/07/2023**
Firma Docente			

INDICE

PRIMERA ENTREGA

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.

Correcciones 25/04:

Luego de la devolución de la primera entrega, en primer lugar se separó la responsabilidad de validar la contraseña en una clase denominada Validador, cuyo método principal es esValida() y utiliza las clases que implementan la interfaz Condición.

Por otro lado, se ha revisado la implementación de los roles que un usuario puede tomar sobre una comunidad, y se ha decidido reducir las clases Administrador y Miembro en una única: Rol.

Se decidió crear una clase y no colocarlas en un enumerado debido a que un usuario (o persona física real) podría tener más de un rol, dependiendo de la comunidad en la que se

tome en cuenta. Un rol tendrá una lista de permisos que luego se utilizarán para, más adelante, corroborar si puede realizar una acción sobre una comunidad.

Esto viene de la mano, a su vez, con la decisión de que la clase Usuario no debería conocer directamente a las comunidades, sino que lo hace indirectamente, a través de un Rol.

En cuanto a los casos de uso, debido a que la relación lineal entre el diagrama de casos de uso y el diagrama de clases no debe ser estrictamente directa, se ha decidido quitar métodos del administrador como *dar de alta*, *dar de baja* y *modificar* en el diagrama de clases.

Por último, quitamos todas aquellas clases que no poseían comportamiento, como *TipoDeTransporte*, *Subterraneo* o *Ferrocarril*.

SEGUNDA ENTREGA

Requerimientos detallados

1. Se debe permitir la administración de entidades.
2. Se debe permitir la administración de establecimientos.
3. Se debe permitir la asignación de personas a servicios de interés.
4. Se debe permitir la asociación de localizaciones a personas.
5. Se debe permitir la asociación de localizaciones a entidades.
6. Se debe permitir la administración de entidades prestadoras y organismos de control.

Administración de entidades

Para esta entrega se amplió el alcance de las entidades del dominio, expandiéndose de servicios públicos de transporte a organizaciones que poseen diferentes sucursales o sedes (por ejemplo, supermercados, bancos, etc.). Se ha cambiado la denominación, entonces, de la clase *ServicioPúblico* para que contemple mayor abstracción a *Entidad*.

Administración de establecimientos

Teniendo en cuenta las nuevas funcionalidades y la ampliación del dominio, se ha creado una clase *Establecimiento*. En ella se comprenden las estaciones y sucursales que sean parte del sistema, de las cuales se tiene, a modo de atributos, su *nombre*, su *ubicacionGeografica* y se posee una lista de servicios ofrecidos (que responden a una clase creada para esta entrega, llamada *Prestación de Servicios*) que recibe el nombre *serviciosBrindados*. Asimismo, se puede consultar por aquellos servicios que presentan problemas, con el método *obtenerServiciosIncidentados*.

Asignación de personas a servicios de interés

Para cumplir con este requerimiento, se implementó en la clase Usuario un método *serviciosDeInterés* (ya que debe ser único por cada persona), el cual a partir de las entidades de interés también asociadas al usuario, obtiene de sus establecimientos los servicios que se encuentren deshabilitados, es decir, que presentan algún incidente.

Se decidió realizar un método y no que se tenga como atributo debido a que no se puede conocer de antemano cuáles son los servicios que presentan algún incidente hasta que se realice el registro pertinente del mismo.

Asociación de localizaciones a personas y entidades

Desde el lado de las personas usuarias de la plataforma, se requiere que tengan una localización asociada para que ellas puedan acceder a aquellos servicios que se prestan en dicha localización. Desde el punto de vista de las entidades, la localización representa los espacios geográficos donde tiene actividad y se ha decidido, entonces que la ubicación geográfica de un establecimiento sea de tipo Localización. Por su parte, las entidades podrán coleccionar las localizaciones de cada establecimiento que tienen asociados con el método *dondeOpera()*.

Una localización puede ser un departamento, un municipio o una provincia. Las localizaciones posibles de ser asignadas deben ser obtenidas del servicio Georef API de la plataforma del Gobierno Nacional. Para esto, se realizó la integración, mediante API REST, contra este servicio.

Se creó una clase Servicio Georef, como clase molde para el modelo JSON, con el patrón Singleton (para asegurarse que en todo el Sistema haya una única instancia de la misma), la cual se asocia a la url de API y contiene los métodos principales para obtener los listados que devuelve el servicio externo. En este caso, los listados que nos interesan son los de departamentos, municipios y provincias.

A su vez, se requiere tener estas clases por separado: una clase por listado y las clases Departamento, Municipio y Provincia. Para lograr abstraer estas tres opciones de Localización y que resulte simple pasar de una a la otra, Localización es una clase abstracta (y no una interfaz ya que los atributos que comparten son *id* y *nombre*) y las tres clases previamente mencionadas heredan de ella. Habrá un método principal que implementen que será obtenerse(), que es llamado por la clase Usuario.

Administración de entidades prestadoras y organismos de control

Para la carga de datos se creó una clase denominada *cargaEntidadesyOrgDeControl*. La misma carga datos de entidades y organismos de control desde un archivo CSV, crea objetos correspondientes a cada entidad y organismo de control, y los imprime.

El mismo posee el siguiente formato:

Organismo de Control, Entidad Prestadora de Servicio

…con la intención de expandirlo para contemplar más información de las Entidades y los servicios que prestan:

Organismo de Control, Entidad Prestadora de Servicio, Entidad, Establecimiento, Servicio

<u>Correcciones 06/06:</u>

- Se modificó la lógica de carga de entidades y organismos de control en el archivo CSV para que no se puedan colocar en forma repetitiva, a través de la validación de si el CUIT ya se encuentra en el archivo o no.
- Se realizó una modificación respecto a la integración con la API GeoRef. Se ha eliminado la clase Localización y se creó otra, llamada Ubicación, permitiendo obtener de la API unas coordenadas geográficas que, a su vez, responden a un municipio que se encuentra en un departamento, que está contenida en una provincia.

TERCERA ENTREGA

Requerimientos detallados

1. Se debe permitir la apertura de incidentes.
2. Se debe permitir el cierre de incidentes.
3. Se debe permitir la consulta de incidentes por estado.
4. Se debe permitir la sugerencia de revisión de incidentes.
5. Se debe permitir el envío de notificaciones a través de Email y WhatsApp.
6. Se debe permitir marcar como “afectado” u “observador” a los miembros de las comunidades para un servicio en particular.
7. Se debe permitir enviar información a entidades prestadoras y organismos de control.
8. Se debe permitir generar los rankings de incidentes.

Apertura de incidentes

Para cumplir con el requerimiento de apertura de incidentes, se implementó inicialmente una clase que abstrae el concepto de "Incidente". Esta clase es creada a través de la clase Singleton *ReportadorDeIncidentes*, que utiliza parámetros como la *prestación de servicio* afectada, el *usuario* que lo efectúa, la *fecha de creación* y la *comunidad* a la que pertenece dicho incidente. A partir de estos parámetros, se genera un incidente que será notificado y agregado a la *Lista de Incidentes* en la clase Comunidad. Esta lista permite que los miembros accedan al listado de incidentes reportados únicos para cada comunidad.

Cierre de incidentes

Para realizar el cierre de un incidente, se implementó un método *cerrarse* en la clase Incidente. Este método, al ser ejecutado, establece la fecha de resolución, cambia el estado de dicho incidente a "resuelto" y finalmente notifica que el incidente ha sido resuelto.

Consulta de incidentes por estado

Los incidentes pueden tener como estado “ACTIVO” o “RESUELTO”, y se resuelven como se detalló en el ítem anterior.
En un futuro, se consultará a la base de datos por los incidentes con un estado en particular. Como de momento no tenemos una, implementamos una clase Singleton RepositorioIncidentes, que almacena los incidentes creados en una lista. Además, agregamos al constructor de la clase Incidente la funcionalidad para que los incidentes creados vayan al repositorio.
En un futuro, RepositorioIncidentes podrá servir como interfaz para interactuar con la capa de datos del sistema.

Sugerencia de revisión de incidentes

La modificación de la localización por parte de un usuario es un evento activador para el Notificador. Cada vez que un usuario modifica su localización, el Notificador detecta incidentes que se han reportado en coordenadas geográficas cercanas a la registrada en la nueva localización de dicho usuario.
De esta forma, se procede a sugerir al usuario vía notificación, siempre que la preferencia del mismo sea de recibirlas en forma instantánea, a que se acerque a revisar si el incidente ya fue resuelto.

Envío de notificaciones a través de Email y WhatsApp

Para el envío de notificaciones a través de Email y WhatsApp se ha tenido que separar el requerimiento en dos partes: por un lado, el medio por el cual el usuario prefiere que le lleguen las notificaciones y por el otro, la periodicidad de las mismas.
En primera instancia, se ha creado a través del patrón Singleton una clase denominada Notificador. Es el componente encargado de notificar a los usuarios interesados lo que ha ocurrido. Es activado por tres requerimientos ya descriptos anteriormente: la creación de un incidente, cuando es marcado como resuelto por otro usuario, o cuando, al modificar su localización, se le sugiere a un usuario revisar manualmente el estado de un incidente.
El método principal de esta clase es *notificar* que recibe como parámetro el Incidente a notificar, consulta los usuarios interesados en el mismo y procede a notificarlos uno a uno recorriendo la lista, según sus preferencias particulares. Para la creación de una notificación, se apoya de la clase NotificacionBuilder que permite instanciar un objeto de la clase Notificacion con distintos textos que aluden a las situaciones comentadas previamente.
Para lograr el envío efectivo de la notificación, se hace uso de un método también denominado notificar, pero que esta vez recibe un usuario específico y la notificación a ser enviada. Del usuario se obtiene su modo de recepción y es a quien se le delega el envío de la notificación. Este modo de recepción es una instancia de la clase Recepción, que

implementa la interfaz que propone la biblioteca Quartz para la implementación de tareas programadas.

El envío de la notificación puede ser en forma sincrónica o asincrónica, teniendo en cuenta los horarios de disponibilidad designados por el usuario. De tratarse del modo sincrónico, se procede a realizar el efectivo envío a través de WhatsApp o correo electrónico, que se detallará luego.

En caso de que el usuario haya elegido horarios para recibir las notificaciones, se tiene que filtrar que la notificación sea únicamente por apertura de incidente y agregarla a una lista de notificaciones sin enviar que se resumirá en el momento correspondiente en una única notificación. El Patrón Command resulta muy útil para diferir la ejecución de estos envíos sin importar el tiempo en el que efectivamente se hagan. Esto se realiza a partir de una tarea programada activada en los horarios de los usuarios, que agrupa las notificaciones pendientes y las resume. El Notificador en este contexto utiliza nuevamente el NotificadorBuilder para agrupar los textos de las notificaciones pendientes y crear una notificación de resumen.

Por otro lado, y haciendo hincapié en los medios de envío de estas notificaciones, para realizarlo a través de correo electrónico se desarrolló la clase Mail, que utiliza ciertas clases que propone Java API Mail. Esta clase posee el método *enviarNotificacionA,* que recibe como parámetros al usuario y a la notificación y lanza una excepción MessagingException en caso de algún problema durante el envío del correo. Este método encapsula la lógica necesaria para enviar una notificación por correo electrónico a un usuario utilizando SMTP (Simple Mail Transfer Protocol) y autenticación.

Se establece el destinatario y remitente del correo electrónico utilizando un método de la clase Message, que son setFrom y setRecipients, los cuales serán directamente obtenidos desde el atributo de email del usuario. Finalmente, se establece el cuerpo del correo colocando el texto de la notificación proporcionada y se envía la notificación por correo electrónico.

Para el envío de notificaciones a través de WhatsApp, al existir más de una API posible para realizar envío de mensajes por WhatsApp, se utilizó el patrón Adapter para implementar el envío de notificaciones.

Se tiene una clase WhatsApp, que tiene de atributo un objeto que implementa la interfaz AdapterWhatsApp. Por esta razón, cuando se tenga que realizar el envío de la notificación a un usuario la responsabilidad recae en la clase concreta que implemente dicha interfaz que se esté utilizando.

Actualmente, la única clase concreta existente es AdapterTwillio, que permite el envío de mensajes de forma muy sencilla y comprensible, con las clases planteadas por la dependencia Twillio.

Marcar como “afectado” u “observador” a los miembros de las comunidades para un servicio en particular

Para la identificación de “afectado/observador” se ha implementado la utilización de un *Enum* denominado *Identificador* con las tres opciones disponibles: *SIN_DEFINIR*, *OBSERVADOR* y *AFECTADO*.

El mismo se trabaja desde la clase Usuario, siendo posible para los mismos el poder definir con qué se identifican en cualquier momento. Este valor es modificable y se encuentra modelado como un Map que conecta una prestación de servicio determinada con alguna de las tres opciones mencionadas.

Ranking de incidentes

Mediante la creación de la clase Ranking, la cual implementa los métodos generarPorPromedioDeCierre y generarPorCantidadDeReportes, se generan los rankings de entidades según el tiempo promedio cierre de incidentes y según la cantidad de incidentes reportados en la semana respectivamente. Estos métodos obtendrán del archivo de incidentes aquellos ocurridos en la última semana para así, evaluarlos según el criterio elegido y obtener las Entidades asociadas a los mismos para realizar finalmente el listado ordenado.

En el método generarPorPromedioDeCierre se utiliza un enfoque escalonado. En primer lugar, se separa el listado de incidentes en una estructura donde se almacenan la Entidad y un listado de incidentes asociadados a esa entidad. Esto se logra mediante el método separarPorEntidadAfectada. Luego, se calcula por cada Entidad el tiempo promedio de cierre según los incidentes correspondientes a la misma mediante el metodo generarPromedioCierrePorEntidad, el cual calculará el promedio mediante la obtención del tiempo de cierre de cada incidente. Finalmente, se generará el ranking de entidades, ordenándolas mediante el uso de la clase RepeticionEntidadesComparador que comparará los promedios asociados a cada entidad y las ordenará de mayor a menor.

Por otro lado, el método generarPorCantidadDeReportes, primero se encarga de filtrar aquellos incidentes que fueron registrados sobre una misma prestaciónDeServicio en menos de 24hs de diferencia y que aún continuan activos. Para ello se utiliza el método filtrarRepetidosEn24hs, que analiza paso a paso la lista de incidentes en todas las cuestiones mencionadas para poder descartar los reportes duplicados. Una vez descartados, se obtienen las entidades asociadas a cada incidente y se realiza un conteo para saber cuántos reportes de incidentes recibió cada una. Finalmente, se genera el ranking ordenado utilizando la clase

de comparación anteriormente mencionada que ordenará las entidades de mayor a menor cantidad de reportes registrados.

Para englobar todos los filtros mencionados (filtrarIncidentesUltimaSemana, filtrarRepetidosEn24hs, separarPorEntidadAfectada otros no mencionados como obtenerIncidentesPorDia, obtenerIncidentesPorPrestacion) se creó la clase Filtrador. Esta clase es la encargada del manejo de los métodos para la modificación de la lista de incidentes, con el fin de lograr un diseño más modular y una mejor organización. Para su utlización en la generación de rankings, se generó una instancia de la misma en la clase Ranking.

Por último, para la generación semanal de los rankings se creó la clase GeneradorRankings que mediante la implementación de la clase Scheduler genera la cronTask necesaria para que los mismos se generen todas las semanas el dia domingo a las 23.59, cuando finaliza el período semanal.

Enviar información a entidades prestadoras y organismos de control

Luego de la generación de rankings semanales, las entidades prestadoras y organismos de control deben poder acceder a la información proporcionada por los mismos. Para ello, se crea un archivo .csv para cada ranking que asocia cada entidad con el valor que le corresponde en base al ranking.