Decisiones de diseño

Casos de Uso:

- Consideramos que los actores son Miembro de comunidad, Administrador de comunidad y Proveedor de plataforma porque son los únicos actores que acceden a las funcionalidades del sistema. Entendemos que hay una entidad Usuario pero lo tomamos como algo interno al sistema y no un actor externo.

- Consideramos que todos los usuarios tienen que iniciar sesión teniendo que registrar su usuario una única vez.

- Consideramos que el proveedor de plataforma no inicia sesión debido a su cargo y puede designar un administrador. La validación de que no tienen conflictos de intereses es ajena al sistema.

Diagrama de Clases:

- Consideramos que los servicios públicos tienen un atributo tipo, que representa si es subte o tren, sólo como etiqueta ya que un subte no tiene diferentes funcionalidades que un tren, o al menos por ahora.

- Consideramos que las comunidades tienen una lista de usuarios que pueden ser miembros o administradores.

- Consideramos que el estado de un servicio representa si está activo o inactivo.

- Decidimos modelar una clase que represente los medios de elevación ya que el enunciado los diferencia de los demás servicios aunque por ahora es todo esquemático porque ningún servicio en particular cumple con funcionalidades.

- La responsabilidad de validar las credenciales es del validador, que no está en el diagrama de clases porque no pertenece al dominio y nadie lo usa, o al menos por ahora.

Algoritmo Validador de Contraseñas:

- La clase Validador es la clase principal que contiene el algoritmo validador de contraseñas.

- Decidimos que las validaciones respetan la interfaz validación, que tiene un método validar. Las validaciones son EsDebil, UsaCredencialPorDefecto y

RespetaPoliticasNIST. Dentro del constructor de EsDebil se lee el archivo que contiene el top 10.000 peores contraseñas para guardarlas en una lista.

- La clase CredencialDeAcceso funciona como un agrupador de todas las cuestiones que el validador debe verificar.

- Consideramos que la creación de una interfaz Condicion, dentro de la clase RespetaPoliticaNIST, es útil ya que nos aporta extensibilidad y mantenibilidad a la hora crear los métodos tieneCaracterEspecial, tieneMayuscula y tieneNumero.

Entrega 2:

Casos de Uso:

- Se agregaron dos actores nuevos, “Entidad Prestadora” y “Organismo de Control”, ambos pueden designar una persona destinada a recibir las problemáticas de los servicios que ofrecen las entidades asociadas.

Diagrama de Clases:

- Implementamos la clase Interés que conoce el Usuario, con el fin de contener la lista de entidades y servicios que le interesan al mismo.
- Por cuestiones de generalización del enunciado, lo que antes se modelaba como Servicio Público ahora pasó a llamarse Entidad, y lo que antes era Estación ahora es Establecimiento.
- Decidimos que la clase Entidad tenga una lista de establecimientos.

Implementación de la carga masiva de datos de entidades prestadoras y organismos de control:

- Decidimos implementar una clase LectorArchivoCSV que incluye un método leerArchivoCSV que actúa de lector genérico recibiendo como parámetro el nombre del archivo, un token y la cantidad de campos.
- Decidimos que la carga de Entidades prestadoras era independiente de la carga de Organismos de control y, por lo tanto, podrían tener un diseño de archivo csv distinto, aunque por ahora no es el caso.
- Cada clase CargaEntidadesPrestadoras y CargaOrganismosControl se encarga de llamar al LectorArchivoCSV definiendo las cuestiones asociadas a su diseño de archivo csv.

Implementación de la integración con el servicio GeoRef API:

- Consideramos que el encargado de realizar las llamadas a la API sería un controller, pero, como todavía no está modelado, no hay nadie que haga llamadas a georef. La integración igualmente está hecha para cuando el controller la llame.

Archivos CSV:

- Decidimos que ambos archivos CSV tengan por el momento un único campo que es el nombre.

Entrega 3:

Implementación en código de los requerimientos:

- **Diseño general de Incidentes:**
  - Dado que cada incidente se considerará válido para la propia comunidad, decidimos que cada comunidad tenga una lista de incidentes.
  - Un miembro puede generar un incidente y se genera en todas las comunidades a las que pertenece
  - Un miembro puede cerrar un incidente dado, de una comunidad dada.
  - Un incidente está asociado a una prestación de servicio que es un value object asociado a una entidad, un establecimiento y un servicio.

- **Apertura de Incidentes:**
  - El incidente lo genera el miembro a través del método de la comunidad generar incidente .
- **Cierre de Incidentes:**
  - El incidente lo cierra el miembro a través del método de la comunidad cerrar incidente, cuando cierra un incidente cambia el estado a false.
- **Diseño general de Notificaciones:**
  - Modelamos las notificaciones simplemente como Strings.
  - La clase encargada de mandar las notificaciones es la clase notificador

- **Envío sincrónico/asincrónico:**
  - Los tiempos configurados son 2: Sin apuros y Cuando sucede.
  - Se usa el patrón Strategy dado que SinApuros y CuandoSucede son estrategias distintas de realizar la misma acción (recibir una notificación).
  - El mensaje recibirNotificacion(miembro, notificacion) es enviado por el notificador.

- **Medios de Notificaciones:**
  - Los medios configurados son 2: WhatsApp (utilizando la API de Twilio) y Email (usando la librería javax.mail)
  - Se usa el patrón Strategy dado que WhatsApp e Email son estrategias distintas de realizar la misma acción (enviar una notificación)
  - El mensaje enviarNotificacion(miembro, notificacion) es enviado por el TiempoConfigurado
  -

- **Eventos que generan Notificaciones:**
    - Apertura de incidente: Se notifica cuando un miembro genera un incidente
    - Cierre de incidente: Se notifica cuando un miembro cierra un incidente
    - Sugerencia de revisión: Se notifica cuando un miembro le envía su localización al sistema y este detecta que está cerca de algún incidente.

- **Diseño general de Afectado/Observador:**
    - Decidimos que el rol temporal por el momento lo pueda modificar el miembro de la comunidad.
    - Dentro de los Ranking tiene más importancia en el ranking 3 que es mayor grado de impacto de las problemáticas.

- **Diseño general de Rankings:**
    - Se usa el patrón Strategy para los criterios de los Rankings ya que son diferentes estrategias para armar un ranking.

- **Generador de Rankings:**
    - Generador De Rankings contiene una lista de incidentes que obtiene de un repositorio de incidentes que contiene todos los incidentes de todas las comunidades, y su método principal es generarSegunCriterio(criterio).

- **Criterio de Rankings:**
    - Diseñamos generar ranking con hashmap que mapea entidades a incidentes, calcula un valor para cada entidad utilizando el método "transformarListaAValor", ordena las entidades según ese valor y devuelve una lista con el ranking resultante.

- **Entidades con mayor promedio de tiempo de cierre de incidentes:**
    - **"tiempoDeCierre"** Calcula la diferencia de tiempo en horas entre la fecha de cierre y la fecha de apertura del incidente utilizando el método "between" de la clase "ChronoUnit".
    - Devuelve el tiempo de cierre en horas como un valor de tipo "double".

      promedio Tiempo Cierre: Este método toma una lista de objetos de tipo "Incidente" como argumento.
      Mapea cada incidente de la lista al tiempo de cierre utilizando el método "tiempoDeCierre" definido anteriormente.
        - Utiliza "mapToDouble" para convertir los tiempos de cierre en una secuencia de valores de tipo "double".
        - Calcula la suma de todos los tiempos de cierre utilizando el método "sum()".
        - Divide la suma de los tiempos de cierre entre el tamaño de la lista de incidentes para obtener el promedio.

- Devuelve el promedio del tiempo de cierre como un valor de tipo "double".

- **Entidades con mayor cantidad de incidentes reportados en la semana:**
  Método "removerSiEstaRepetido":
  toma tres argumentos: un objeto de tipo "Incidente" llamado "incidente", una lista de objetos de tipo "Incidente" llamada "incidentes" y un objeto de tipo "Incidente" llamado "incidenteContraElQueComparo".
  Verifica si el "incidente" está repetido dentro de un cierto plazo en relación al "incidenteContraElQueComparo" utilizando el método "estaRepetidoDentroDelPlazo" de la clase "Incidente".
  Si el "incidente" está repetido, se elimina el "incidenteContraElQueComparo" de la lista "incidentes".

  Método "eliminarRepetidoEn24hs":
  Toma tres argumentos: un objeto de tipo "Incidente" llamado "incidente", una lista de objetos de tipo "Incidente" llamada "incidentes" y un entero llamado "posAPartirDelIncidente".
  Utiliza el método "subList" de la lista "incidentes" para obtener una sublista a partir de la posición "posAPartirDelIncidente" hasta el final de la lista.
  Utiliza el método "forEach" de la sublista para iterar sobre cada "Incidente" y llama al método "removerSiEstaRepetido" para eliminar los incidentes repetidos dentro de un plazo de 24 horas.

  Método "filtrarRepetidos":
  Toma una lista de objetos de tipo "Incidente" llamada "incidentes".
  Utiliza el método "forEach" de la lista "incidentes" para iterar sobre cada "Incidente".
  Llama al método "eliminarRepetidoEn24hs" pasando el "Incidente" actual, la lista "incidentes" y el índice del "Incidente" en la lista.

  - Diseñamos métodos para eliminar incidentes repetidos dentro de un plazo de 24 horas en una lista de incidentes y transformar la lista resultante en un valor, que es la cantidad de incidentes restantes.
- **Mayor grado de impacto de las problemáticas:**
  - Diseñamos solo el prototipo ya que el enunciado no lo especifica detalladamente.