**Justificaciones de las decisiones de diseño tomadas**

* Primera entrega: Modelado en Objetos Parte I: Puesto a punto del entorno de desarrollo y primera iteración del diseño

En primer lugar, fuimos identificando y guardando información relevante relacionada al Estado Interno de cada clase, la cual consideramos que vamos a necesitar en un futuro para desarrollar el sistema.

En Servicios Públicos, creamos una sola clase que represente ferrocarriles y subterráneos porque no identificamos nada que los diferencie a la hora de utilizarlos en el sistema en base a la funcionalidad.

En Servicios, separamos los Medios de Elevación de los Baños ya que, tenemos que guardar información propia de cada una de esas clases. También diseñamos una superclase abstracta llamada Servicio para poder tratar a ambas clases de manera polimórfica y guardar el atributo que tienen en común (Estado). Por esto mismo, decidimos crear una clase abstracta y no una interfaz.

En Comunidades y Usuarios, decidimos crear una lista de Administradores como atributo perteneciente a la clase Comunidad porque si nosotros le asignamos un rol (como estado interno) a la clase Usuario íbamos a tener muchos problemas ya que hay usuarios que pueden tener un rol distinto según la comunidad a la que pertenezcan. Además esto mismo genera tener una relación de asociación doble, lo cual conlleva a tener más acoplamiento entre esas dos clases.

Utilizamos una relación de agregación entre las clases Comunidad y Miembro, ya que las comunidades necesitan a un miembro para existir, creemos que no debe haber comunidades sin miembros, pero los miembros pueden existir sin necesidad de pertenecer a una comunidad.

Estas dos clases, por el momento, solo contienen atributos debido a que no identificamos ninguna funcionalidad (método) necesaria para los requerimientos dados de esta entrega.

Para verificar si una contraseña es válida, creamos una clase Verificador la cual conoce y utiliza distintas clases de condición (Longitud, Credencial y Top 10000). Estas condiciones se basan en el patrón Strategy, donde consideramos cada condición como una estrategia, ya que sino cada vez que queramos agregar una nueva condición deberíamos modificar nuestra clase Verificador la cual tendría toda la responsabilidad, generando poca cohesión ya que se encargaría de hacer muchas cosas.

* Segunda entrega: Modelado en Objetos Parte II e incremento de funcionalidades:

Dado que para la presente entrega se agregaron las organizaciones que poseen sucursales a los servicios a monitorear, estos mismos tuvieron que ser adaptados y generalizados cambiando el modelo para poder cumplir con la funcionalidad del sistema.

Por eso, nuestras clases “Servicio Público“ y “Estación“ se vieron afectadas. Se dividió por un lado, las líneas de servicio de transporte y las organizaciones en una clase llamada “Entidades”, y las estaciones y sucursales en otra llamada “Establecimientos”, cada una con sus respectivos atributos.

Por otro lado, la clase “Establecimientos” se relaciona con una interfaz “Localización”, la cual es implementada por las clases Municipio y Provincia, con el objetivo de establecer una ubicación para cada uno de los establecimientos y así saber dónde presta sus servicios la Entidad. Esto lo logramos mediante la implementación de una API REST que proporciona esa información a través de la plataforma de datos abiertos del Gobierno Nacional Argentino. Además, la clase Usuario también se relaciona con dicha interfaz dado que los usuarios del sistema tienen interés en acceder a los servicios que se prestan en la localización en que se encuentran.

En la clase Importador, la cual se encargará de la carga de datos de entidades prestadoras y organismos de control a través de un archivo csv, decidimos implementar el Patrón Strategy, para generar mayor cohesión y mantenibilidad en esta clase a lo largo del tiempo. Por eso, creamos dos clases que corresponden a las estrategias de importación, tanto para las entidades prestadoras y los organismos de control, que implementan una interfaz EstrategiaImportación, que se relaciona con la clase Importadora.

Para la clase “Entidad” le agregamos una lista de Atributo Especiales como atributo para poder tener atributos específicos de cada tipo de entidad (línea de transporte o una organización). Tuvimos la necesidad de hacer esto, ya que la “Entidad” representa tanto servicios de transporte como diferentes organizaciones, pero aunque se comportan de la misma manera, cada uno tiene sus propios atributos (distintos a los de los demás).

Si nosotros queremos modelar una línea de servicio públicos, que cuenta por ejemplo con una lista de estaciones no tendría sentido que un banco tenga estos atributos, así que como solución se propone modelar una clase abstracta “AtributoEspecial” de la cual van a heredar diferentes clases tales como “AtributoInt”, “AtributoEstablecimiento”, “AtributoString”, etc. dependiendo del tipo de atributo que se quiera representar se va a relacionar con la entidad definida.

Por ejemplo, para una línea de trenes teniendo una lista de estaciones, en la primera entrega, si solo cambiamos el nombre de Línea por Entidad, iba a seguir teniendo esas estaciones que si la entidad llegaba a ser un banco, no tendría sentido que estén esos atributos.

* Tercera entrega: Modelado en Objetos Parte III e incremento de funcionalidades

La siguiente entrega comprende el manejo de los incidentes, es decir su apertura y cierre, el ranking de los mismos y el envio de notificaciones.

Los miembros de una comunidad pueden reportar o cerrar incidentes, haciendo un llamado a la comunidad que crea o resuelve un incidente, es decir una instancia de clase que contiene el servicio que se encuentra afectado, el usuario que lo reportó, la hora de su reporte, su estado, entre otros. Un incidente implica un cambio de estado en el Servicio, que pasa a estar *FUERA DE SERVICIO.*

Asimismo para cerrar incidentes, sobre la misma instancia del incidente se cambia el estado del mismo a *RESUELTO* y se establece su hora de cierre, ya que la necesitaremos posteriormente para elaborar los rankings. En este caso el estado del servicio para a estar *EN FUNCIONAMIENTO*.

Nuestra clase servicio ahora pasó a ser una clase abstracta ya que ahora surgen dos abstracciones que son los servicios simples y los servicios compuestos, y dado que los mismos pueden cambiar varias veces de estado nos puede resultar útil contar con esta información de cambios, para ello elaboramos un historial de cambios, que es una lista de CambiosDeEstado que contiene el nuevo estado, el usuario que realizó la acción y la hora.

Las acciones sobre los incidentes, ya sea su apertura o cierre deben ser notificados a los miembros de la comunidad. Para abordar esto contamos con un notificador que usa una estrategia de notificación que cada usuario tiene, esta estrategia se asocia a dos interfaces una es el medio de notificación que puede ser implementado por dos clases, NotificarWhatsApp y NotificarMail, la otra es el tipo de notificación que puede ser sincrónica o calendarizada. Al diseñarlo de la siguiente manera dejamos la posibilidades de que surjan por ejemplo nuevos medios de notificación que si en futuro requieren ser añadidos al presente sistema solo bastará con diseñarlos y que implemente la interfaz.

En resumen el usuario configura su estrategia de notificación eligiendo el medio y tipo, y cuando sucede un hecho que requiere un aviso mediante una notificación, el envío de la misma se realiza teniendo en cuenta la configuración elegida.

Algo similar sucede con las sugerencias de revisiones, solo que en este caso se omite el tipo de notificación elegida, ya que la misma debe ser enviada cuando el usuario se encuentra cerca, pero se sigue respetando el medio elegido.

Decidimos implementar el Patrón Strategy para calcular los tipos de ranking. Usamos una interfaz llamada EstrategiaRanking y tres que calculan los rankings con cada algoritmo distinto. Y para generar la lista de los tres rankings, utilizamos la clase RankingSemanal que va a calcular los tres rankings para cada entidad.

A la hora de abarcar el punto de los usuarios de servicios y observadores, en la clase Comunidad pusimos como atributos, dos listas, una de miembros afectados y otra de observadores, para poder diferenciarlos del otro. Respecto a los roles de administrador y miembro común, como en la entrega anterior, la comunidad tiene una lista de miembros y, cada miembro tiene como atributo el rol que le corresponde.

* Cuarta Entrega: Servicio y Persistencia de datos

Para esta entrega se solicita la persistencia de ciertas entidades desarrolladas en las entregas anteriores y el diseño e implementación de un Servicio, en este caso se nos asignó el Servicio N°1.

La primera parte, la de persistir los datos, utilizamos el framework de mapeo-objeto relacional Hibernate visto en clase. El mismo nos permite realizar anotaciones para indicar que entidades queremos persistir, de qué manera, y en caso de necesitarlo especificar las relaciones entre las mismas. La implementación se puede ver detalladamente en el código y en el diagrama entidad relación (DER).

Con respecto al Servicio asignado, el Servicio N°1, que tiene como objetivo sugerir la fusión de comunidades, y para esto analizará las comunidades existentes y buscará diferentes coincidencias entre las mismas. El diseño del mismo se detalla a continuación.

Implementamos clase Receptor que se va a encargar de generar las propuestas y realizar la verificación de los criterios para evaluar si las comunidades cumplen el o los requisitos para ser fusionadas. La propuesta también es una entidad que contiene dos candidatos (comunidades) a fusionar. Por último contamos con lla clase Fusionador, que como bien dice su nombre será la encargada de emplear la fusión de dos comunidades, juntándolas en otra más grande y retornarla.

Para manejar los criterios para la fusión creamos diferentes clases que implementan un mismo metodo de una interfaz *cumpleCriterio*(2), estas clases son las que se van a encargar de manejar la lógica para determinar si las comunidades pueden ser fusionadas, esta decisión además nos otorga extensibilidad, ya que si queremos agregar otro criterio agregamos otra clase que implemente la interfaz y implemente la logica de verificaion ella misma.Si nos fijamos bien es un diseño pensado sobre patrón strategy.