Decisiones de diseño:

Decidimos crear una clase Usuario, donde el mismo pertenece a una o varias Comunidades (compuestas por varios Usuarios), además de tener atributos identificadores como nombre, apellido y correo electrónico. También la clase Usuario tiene asociada una clase Credencial, más que nada para agrupar atributos y por un tema de normalización de datos, donde esta, va a tener un usuario y su contraseña (ya que de esta forma favorecemos el desacoplamiento) con su respectiva fecha y cadena. Decidimos que Contraseña sea una clase ya que cuando desarrollamos el validador de contraseñas, nos dimos cuenta que necesitábamos guardar la fecha en la que esta era instanciada.

Decidimos que un Usuario tenga un atributo rol, ya que de esta forma, se va a permitir en entregas posteriores, manejar permisos para permitir realizar o no ciertas acciones. Es algo que no se especifica en el enunciado, pero lo pensamos desde el lado de alguien que utiliza el sistema.
Por otro lado decidimos no usar herencia para Administrador y Miembro ya que esto nos limitaría mucho el diseño, y anularía la posibilidad de que un miembro sea a su vez administrador.

Se menciona que los usuarios pueden compartir información sobre los servicios. Para esto, consideramos que la mejor forma, es que los usuarios cuando comparten información (a través de un “botón”), esten creando una solicitud para cambiar el estado de un servicio. Para que se modifique el estado de un servicio por una solicitud debe ser aprobada dicha solicitud. Y luego de analizar las distintas opciones llegamos a la conclusión que lo mejor es una solución híbrida donde se puedan aceptar de dos maneras: puede ser porque fue aprobada por un administrador o, también puede ser aprobada automáticamente porque llegaron cierta cantidad de solicitudes iguales sobre el mismo servicio. Lo pensamos haciendo de cuenta que estábamos usando la aplicación y lo comparamos con la aplicación Waze (una app al estilo google maps en la que los usuarios informan en tiempo real accidentes, tráfico, controles, etc). Los usuarios pueden informar cosas, y después de que otros usuarios "confirman/informan lo mismo", se hace el cambio

Analizamos varias opciones: (1) Que los usuarios pudieran modificar el estado de un servicio directamente, (2) que los usuarios puedan crear una solicitud, que deba ser solicitada por un número definido previamente para luego cambiar el estado del servicio automáticamente y (3) que la solicitud sea aceptada por un administrador.

Sobre (1) trae varios problemas, entre ellos:

1. Los usuarios podrían estar libremente cambiando el estado de un servicio, pudiendo generar información inconsistente ante muchos cambios consecutivos.
2. Si los usuarios pueden acceder al estado del servicio estamos rompiendo el encapsulamiento.

Sobre (2). En caso de que solo se implemente esta opción: Digamos que un usuario crea una nueva solicitud para cambiar el estado de un servicio, y el valor predefinido para cambiar el estado es de 100. Hasta que no haya otras 99 personas que quieran cambiar el servicio, muchas otras personas que revisen el estado del servicio verán que está disponible cuando la realidad es otra, perjudicando así a los usuarios.

Por último, sobre (3), en caso de implementarlo como única opción, ocurriría que tendríamos que tener a un administrador 24/7 pendiente de las solicitudes, y justamente no es el objetivo del sistema.
Por eso concluimos que una solución intermedia, entre (2) y (3) sería lo ideal.

No implementamos ningún método o algo relacionado a “compartir información”, “aceptar solicitud”, o cosas por el estilo ya que creemos que es algo relacionado a los roles que cumplan cada usuario y es algo que lo van a hacer físicamente por decirlo de alguna manera o apretando un botón, que esto a su vez va a disparar acciones, pero que no necesariamente deben estar implementadas en la clase usuario o comunidad.
El mapa o camino que hicimos en nuestra cabeza sería algo así como:

El usuario apreta botón de compartir información con el servicio afectado y si está disponible o no —> se crea una solicitud con dichos datos —> se agrega a la lista de solicitudes de la comunidad —> un administrador debe aceptarla o rechazarla (con otro botón) o pueden ser aceptadas si se llega a cierta cantidad de solicitudes iguales sobre el mismo servicio.

En cuanto a la entidad ServicioPublico, tomamos la decisión de que sea una interfaz ya que tuvimos en cuenta la posibilidad de que en el futuro se agreguen más servicios públicos, y es probable que todos cumplan cierto comportamiento, de esta manera estaríamos pensando en la extensibilidad del sistema.

Respecto a la entidad TransportePublico, definimos que tendrá un tipo, que será un enum. Este enum, cuenta con dos posibles valores (SUBTE y TREN). Lo tomamos así ya que, hasta donde llega el enunciado actual, no parece modificar el comportamiento del TransportePublico, pero sí especifica que puede ser uno de estos dos.

Finalmente el estado de los servicios decidimos modelarlo como una variable booleana, dado que de momento los únicos estados posibles son disponible o no disponible, por lo que consideramos que generar una entidad Estado, o darle mayor profundidad, parece estar de más.

En este momento la ubicación geográfica de la Estación decidimos ponerlo como String, pero más adelante quizá sea otra cosa. Puede que debamos utilizar una API.

SOBRE EL VALIDADOR DE CONTRASEÑAS:
Nuestro diseño fue pensado priorizando la extensibilidad de validador, es por eso que decidimos tener una clase ValidadorDeContrasenia y que tenga como atributo una lista con todas las Restriccion(es) que debe cumplir. Entonces, cada vez que se deseen agregar/quitar restricciones simplemente habría que modificar esa lista y crear esa nueva restricción. Una Restriccion, es una interfaz que tiene la firma esValida(contraseña). Es implementada por los distintos tipos de restricciones, como por ejemplo una ExpresionRegular, Caducidad, Longitud, ListaDeContraseniasDebiles y sus ramificaciones.
En el código hay algunos comentarios que nos parecieron importantes aclarar.
El manejo de errores, utilizamos excepciones, decidimos crear una personalizada llamada ContraseniaInvalidaException. En caso de que una contraseña no sea válida, se informará de cada una de las restricciones que no se cumplan. Para poder informar cada uno de los

mensajes, en el método esValida(contraseña) del validador, recorremos la lista con un bucle for, y en caso de que no se cumplan restricciones se van guardando en una lista para luego informarlas todas juntas con una excepción, para esto, cada restricción implementa el método getMensajeError() que retorna un mensaje para cada tipo de restricción.
La contraseña, como se mencionó anteriormente, decidimos que sea una clase, ya que debe guardar con ella la fecha en que fue instanciada.