Decisiones de la entrega N° 1:

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.

Decisiones de la entrega N°2:

Respecto a la ampliación del dominio.
Puesto que el enunciado amplía el dominio para esta entrega, decidimos abstraer lo que antes consideramos como TransportePublico a una clase Entidad y sus Estaciones, como Establecimientos. Además tampoco hace falta considerar las estaciones de origen y fin, que en la entrega previa hacían falta.
Con esta nueva abstracción pueden considerarse distintos tipos de Entidades y sus Establecimientos, y no solo centrarse en aquellas entidades respectivas del transporte público.

Ya que el enunciado nos habla de entidades prestadoras y organismos de control, decidimos modelar ambas como clases (“OrganismosDeControl” y “EmpresaPrestadoraDeServicios”). Si bien todavía las mismas no tienen un comportamiento bien definido más allá del de los casos de uso, decidimos traer ambas entidades para esta entrega ya que consideramos que son necesarias para llevar a cabo la implementación de la lectura del archivo CSV.

Dado que una empresa puede designar a una persona a la cual le llegará información acerca de las problemáticas de los servicios que ofrece la misma, se decidió agregar un nuevo rol para un usuario, el mismo será “RESPONSABLE” y representará a las personas que son designadas por cada empresa para cumplir dicho rol. Por el momento decidimos dejarlo como rol ya que, por el momento, no sabemos de qué manera se va a compartir la información y el comportamiento que va a tener este responsable.

Respecto a los intereses de los usuarios.
Decidimos que cada Miembro tenga una lista de Intereses como atributo, donde cada Interés es una clase donde tiene como atributo la Entidad y el Servicio que este contempla. Lo pensamos de esta forma así favorecemos el hecho de agregar o quitar Intereses a nuestro miembro, ya que consideramos que esto puede ir variando en nuestro sistema a lo largo del tiempo. También consideramos agregar a la clase miembro una localización de interés, donde cumpliría la función que el enunciado solicita, que sería la de tener interés sobre todos los servicios que se prestan en esa localización. Este atributo es de tipo localización y puede ser tanto un Municipio, una Provincia o un Departamento.

Respecto a la localización que pueden tener tanto los usuarios como las entidades.
Para las localizaciones decidimos hacer una clase abstracta **Localización** con los atributos **ID** y **nombre**, que son los que va a traer a través del **ServicioGeoref**. La clase abstracta decidimos diagramarla debido a que como la localización puede ser tanto un **Municipio**, **Provincia** o **Localidad**, estas puedan usarse de forma polimórfica compartiendo comportamiento y atributos.
Además consideramos que los Establecimientos de una Entidad serán aquellos que tengan una localización, dado que son una abstracción de un lugar físico.

Respecto a la implementación de la lectura de datos por archivo CSV.
En el caso del **LectorDeArchivosCSV** para la carga masiva de empresas y organismos de control, decidimos que trabaje con dos archivos distintos, uno para empresas y otro para organismos. Esto lo decidimos de esta manera para evitar validaciones sin sentido y que los métodos de carga sean lo más cohesivos posibles. Además, en caso de que se requiera, que empresas y entidades no necesariamente reciban los mismos atributos.
En cuanto al lector, decidimos que por el momento tenga una lista de empresas y una lista de organismos, que cuando se ejecuten los métodos de carga se van a llenar estas listas, sin embargo esto podría modificarse y hacer que los métodos directamente retornen las empresas y organismos, esto va a depender de la forma en que lo usemos en un futuro.
Para el archivo CSV por el momento hicimos que las empresas y organismos solo tengan un atributo nombre, pero se podrían agregar más atributos de forma sencilla haciendo que el constructor de las clases reciban más parámetros y ya.

Decisiones de la entrega N°3:

El establecimiento ahora conoce la entidad a la que pertenece ya que necesitamos acceder a ese dato para poder generar uno de los rankings.

Los miembros solo pueden tener una comunidad
Esto es así porque de otra manera el miembro podría tener un rol distinto para cada comunidad (observador o afectado), teniendo que generar clases intermedias que consideren este caso resultando en una solución muy compleja

Los estados de los incidentes representan lo mismo que el estado de un servicio
Dado que se considera que si existe una incidencia sobre un servicio y esta se encuentra abierta, el servicio ya no se puede operar con normalidad, decidimos interpretar el estado del mismo, a través de la incidencia.

Las prestaciones de servicio ahora tienen un establecimiento
Esto lo decidimos dado que necesitábamos conocer la localización de un incidente. Para ello consultaremos la localización del establecimiento en el que se encuentra dicho incidente.

Las notificaciones se hacen a través de una clase Notificador que tiene la responsabilidad de revisar el medio y la estrategia de notificación del miembro y según eso notifica de esa forma. El notificador será instanciado en la capa de controladores y debería poder enviar una notificación de manera sincrónica o asincrónica ante un determinado evento.

Las clases MailSender y whatsappSender se integran con javaxMail y twillio respectivamente y serán utilizadas por el notificador como medio de envío para las notificaciones.

Creamos una interfaz común llamada Notificable para los distintos tipos de notificaciones. Esta interfaz común es la que implementaran los distintos eventos que tenemos hasta el momento, y tiene los métodos para construir un mensaje o un asunto dado el tipo de notificación.

Las notificaciones son 3 clases, que representan los eventos por los cuales el notificador envía la notificación a los miembros. Estas son: “NuevoIncidente”, “RevisarIncidente”, “CierreIncidente”. Son clases que tienen la responsabilidad de construir un mensaje y dárselo al notificador para que este se lo envíe de una manera u otra al receptor.

Al plantear las notificaciones de esta manera, implementando una interfaz común, ganamos en testeabilidad, cohesión, pues la responsabilidad de los distintos tipos de mensajes está en cada una y además delegamos responsabilidades que podrían tener erróneamente otra clase.

Las notificaciones sincrónicas se envían en el momento que sucede un evento.
Al abrir un nuevo incidente y este ser agregado en la comunidad del miembro notificante, se notificará a cada miembro de dicha comunidad.
Al cerrar un incidente, se notificará dicho cierre a cada miembro de la comunidad del miembro que notifica el cierre.
Y en caso de que un miembro cambie su localización, el sistema podría detectar que se encuentra cerca de un incidente, y en cuyo casa se le solicitará que vaya a revisarlo

Tomamos como que un miembro está cerca de la localización de un incidente, cuando la localización del miembro es literalmente la misma que la del incidente, esto lo hicimos ya que a fines prácticos simplifica muchas cosas. Si no, deberíamos meternos con coordenadas, distancias mínimas y cosas que no están tan buenas.

Los incidentes a la hora de reportarse, pueden tener una observación, que en este caso nosotros la implementamos como un atributo “descripción”, que sería la “general” del incidente, por decirlo de alguna manera. Por otro lado, tenemos lo que son las observaciones que realizan otros miembros sobre un incidente ya abierto. Esto no es más que un atributo, una lista de Observación (una clase que tiene un notificante, una descripción y una fecha), que se le asigna a ésta en forma de clase con el principal objetivo de no perder trazabilidad y formar una especie de historial de versiones de las distintas etapas por las que pasó un Incidente.

Decisiones de la entrega N°4:

Persistencia:

Las entidades que decidimos persistir fueron:

- Comunidad
- Miembro
- Usuario
- Incidente
- Observación
- Localización
- Entidad
- TipoEntidad
- Establecimiento
- Servicio
- PrestacionDeServicio
- Interés

El resto de entidades que son utilizadas a nivel de objetos, decidimos no persistirlas ya que, son entidades que no interesa guardarlas o bien son encargadas del comportamiento del sistema, como puede ser el Notificador o el Exportador

Respecto al uso de hibernate y annotations en las relaciones OneToMany o ManyToOne lo hicimos de la siguiente manera

Por ejemplo, del lado de la Entidad (que tiene una lista de establecimientos):

```
@Getter
@OneToMany(mappedBy = "entidad")
private List<Establecimiento> establecimientos;
```

Del lado del establecimiento:

```
@Getter @Setter
@ManyToOne
@JoinColumn(name = "id_entidad", referencedColumnName = "id")
private Entidad entidad;
```

Notar que en el Establecimiento, debíamos agregar como atributo la Entidad a la que dicho establecimiento perteneciera, para así poder mapear la relación de esta manera.

La otra forma de mapeo hubiera sido desde la Entidad, “ponerle” una columna para el id_entidad al Establecimiento, pero nos parecía que no era responsabilidad de la Entidad tener que mapear una columna de otra clase que no fuera ella.

Respecto a relaciones ManyToMany

En nuestro caso solo tuvimos una, y se generó a partir de la lista de PrestacionDeServicio que debimos agregar a la comunidad, como “ServiciosParticularesObservados”. La tabla en cuestión que se genera como intermedia de Comunidad y PrestacionDeServicio es: “comunidad_prestacion_de_servicio” que tiene una referencia a una Comunidad, y una PrestacionDeServicio.

Respecto a Impedance Mismatch resueltos

- Respecto a la identidad, agregamos un “private int id” como clave subrogada en cada entidad que nosotros consideramos persistente.
- Respecto a los tipos de datos.
    - Para aquellos datos que hibernate no mapea de forma automática, usamos distintos “AttributeConverter”. LocalDate lo mapeamos como Date. Y LocalDateTime lo mapeamos como Timestamp.
    - Para los enumerados, usamos la annotation `@Enumerated(EnumType.STRING)` De esta forma hibernate toma los valores del enum como String y los mapea como un Varchar(255)
- Respecto a la herencia. En nuestro caso solo teníamos una herencia y era en la Localización. Decidimos mapearla usando la estrategia de “Single Table”, la cual cuenta con los siguientes campos: Id, nombre, es_provincia, id_provincia y discriminador.
    - El campo discriminador es el que nos permitirá diferenciar si se trata de un municipio, un departamento o una provincia.
    - El campo es_provincia, en realidad es un atributo que agregamos para facilitarnos el analisis de si un miembro estaba cerca de un incidente al cambiar su localización.
    - El campo id_provincia, referencia a una provincia (fila de esta misma tabla), y estará seteado solo para los municipios y departamentos. En caso de ser la localización una provincia, estará en null.
    - Ej:

id	nombre	es_provincia	id_provincia	discriminador
1	Buenos Aires	1	null	Provincia
2	Tandil	0	1	Departamento
3	Jujy	1	null	Provincia
4	Pinamar	0	1	Municipio
5	General Juan Madariaga	0	1	Departamento

Utilizacion del Patron Repositorio

Para esta etapa de persistencia fue necesario el modelado y la implementación de entidades repositorios para proporcionar una abstracción de alto nivel entre la lógica del monitoreo y el acceso a la base de datos. Para ello se utilizó el Patrón Repository creando operaciones básicas como agregar (persistir), eliminar, modificar, buscar, entre otras particulares de cada entidad.

Se decidió crear su respectivo repositorio de las siguientes 13 entidades; Comunidades, Establecimientos, Incidentes, Intereses, Localizaciones, Miembros, Notificaciones, Observaciones, PrestacionesDeServicios, Servicios, TipoEntidades, Usuarios y por último Entidades.