Cuidándonos

PUNTO 1: Arquitectura

1.

A. Caso **cola de mensajes**:
   - Mantenibilidad: Puede ser que el sistema esté muy acoplado a el firebase, por lo que sería un problema a la hora de cambiar de plataforma. Este software de uso libre de Google realiza el envío de mensajes y la obtención de datos. Se pueden adoptar prácticas de diseño que separen las responsabilidades y reduzcan el acoplamiento, lo que permitiría una migración más sencilla si fuera necesario cambiar de plataforma en el futuro.
   - Disponibilidad: El envío y recepción de mensajes es mucho más eficiente que en el caso b, ya que se maneja con firebase compartiendo los datos en tiempo real. La dependencia de llamadas periódicas al servidor central implica que la disponibilidad de datos actualizados está limitada por la frecuencia de estas llamadas.

B. Caso **proceso propio**:
   - Mantenibilidad: Proporciona un mejor desacoplamiento que facilita los posibles cambios en el sistema. Además, la abstracción en la comunicación entre los componentes del sistema proporciona una estructura más flexible, lo que facilita el mantenimiento y la escalabilidad.
   - Disponibilidad: La comunicación a través de Firebase es eficiente y en tiempo real, lo que garantiza una disponibilidad casi instantánea de los datos.

2.

a.

- ☐ La aplicación deberá ser nativa en su totalidad.
- ☑ La aplicación deberá tener una capa de visualización que corra sobre el sistema operativo del smartphone y la lógica de negocio implementada en un servidor en la nube.(1)
- ☐ El cálculo de la distancia deberá hacerla la propia aplicación.
- ☑ El cálculo de la distancia deberá ser delegada a un componente de terceros (tal cual lo presenta el dominio actualmente).(2)
- ☑ El dominio podría ser implementado en una base de datos no relacional.(3)
- ☑ La aplicación podría tener persistencia políglota(4)

b. Justifique sus elecciones del ítem anterior según el atributo de calidad *performance* y cualquiera de los siguientes que elija que crea que aplican al dominio:

- ☐ Seguridad
- ☑ Portabilidad
- ☑ Funcionalidad
- ☑ Usabilidad
- ☑ Eficiencia
- ☐ Madurez

1) Esta elección la podemos justificar a partir del atributo de calidad “Portabilidad” ya que esto permite que sea transferido de forma efectiva, en este caso, de un sistema operativo a otro. Y nos brinda también usabilidad, ya que muchos más usuarios podrán acceder a la app sin inconvenientes además de que esta es clara en su uso.
2) Según performance lo que nos brinda es un menor uso de nuestros recursos tanto de sw, como horas persona.
3) Esta elección la podemos justificar mediante el atributo “Eficiencia”, ya que, las bases de datos no relacionales pueden mejorar la eficiencia de la aplicación en términos de rendimiento y escalabilidad.
4) Esta elección también la podemos justificar mediante el atributo de diseño “Portabilidad”. La persistencia políglota permite que la aplicación pueda utilizar múltiples tecnologías de almacenamiento de datos según las necesidades específicas de cada parte de la aplicación.

PUNTO 2: Modelo de Diseño

Decisiones de Diseño:

1) Modelar Usuario:

   - Una opción era modelar al Transeúnte y al Cuidador como clases que heredan de una clase Usuario pero esto no permite que los tipos cambien, es decir, que un transeúnte sea un cuidador y viceversa. Además evaluamos esta opción por si posteriormente ambos tenían comportamientos distintos.
   - Otra opción era contemplar esta característica por medio de un enum. Pero, leyendo el enunciado vimos que ninguna de las opciones era de gran utilidad ni necesaria, ya que simplemente el viaje es el que debe tener registro de qué tipo de usuario son los que intervienen. Por esto, modelamos al Usuario como una clase con sus respectivos atributos.

2) Modelar ReaccionIncidente:

   - En el atributo reacción, en la clase Usuario, utilizamos el patrón Strategy ya que el mismo podrá ser cambiado en momento de ejecución y cada tipo de reacción debe alertar de manera distinta, es decir, tiene un comportamiento distinto.

3) Modelar Viaje:
   - Como el Viaje debe llevar registro de quienes son los cuidadores y quién es el usuario cuidado. Realizamos una lista de cuidadores a la cual se le agregan estos usuarios cuando ellos acepten la solicitud.

4) Modelar InfoViaje
   - Esto se diseñó para una posterior carga en la implementación de reacciones. Dependiendo si distintas reacciones utilizan los mismos o diferentes datos del viaje/usuarios se agreguen al InfoViaje sin mayores complicaciones, procurando extensibilidad y mantenibilidad.

5) Modelar Dirección:

   - Decidimos modelar la dirección como una clase, lo que nos permite gestionar de manera más efectiva sus atributos y operaciones relacionadas. El atributo barrio lo modelamos como un enum. Esta elección nos permite evitar inconsistencias al limitar las opciones a un conjunto definido y predefinido de valores.

2.

```
public class Direccion{
      .
      Time tiempoEspera = 0;
}
```

```
public class Usuario{
EleccionSobreParadas eleccion;
}
```

```
public class Viaje{
List<Direccion> paradas;
}
```

```
public class EsperaNMinutos{

tiempoViaje(paradas : List<Direccion>){
      calculadora = new CalculadoraDeTiempo()
      List<Time> tiempos = new List<Time>

      Time timpoParcial;
       for(int i=0 ; paradas.size() -1 > i+1 ; i++){
      paradaInicio = parada.get(i);
      paradaFinal = parada.get(i+1);
      Time tiempoParcial += calcularTiempo(paradaInicio, paradaFinal);
 }
       Time tiempoTotal = tiempoParcial +
       paradas.sum{unaParada => unaParada.tiempoEspera()}

       return tiempos.add(tiempoTotal );
}
```

```
public class AvisaEstado{

tiempoViaje(paradas : List<Direccion>){
      calculadora = new CalculadoraDeTiempo()
      List<Time> tiempos = new List<Time>
      cant = paradas.size()
      int j = 0;
      for(i = 1;cant > i; i++){
      unTiempo = calculadora.calcularTiempo(parada.get(i), parada.get(j));
      j++
      tiempos.add(unTiempo);
}
return tiempos }}}
```