## PUNTO 1

### Persona

Modelamos una clase Persona para transeúntes y para cuidadores, de forma que una persona pueda tener comportamiento de Cuidador o de Transeúnte dependiendo de los valores que se instancian.
El crear una clase abstracta "Persona" y que la hereden unas clases "PersonaTranseunte" y "PersonaCuidadora" sería un grave error ya que no se podría cambiar la naturaleza de una instancia, por ejemplo “cuidador” a “transeúnte”. Si se podría si se elimina el objeto y se vuelve a crear con las propiedades de la otra clase (mala práctica).

En cuánto a los atributos:

-Tiene sentido que en común estén el nombre, el apellido, la dirección (que se abstrae como una clase), la fecha de Nacimiento (aunque la consigna diga que se pide la edad, este es valor es CALCULABLE, si pusiéramos un atributo "edad", habría que cambiarlo todos los años, en vez de eso, creamos el método edad() que nos retorna la edad real en todo momento).

- La forma de reaccionar a un accidente tiene que estar especificado en la clase porque es propio de cada persona y puede cambiarla si quisiera, el sexo tiene que ser un enumerado (no un string) por si se quieren añadir otros, y el usuario (al estar hablando en un contexto de una aplicación tiene sentido que este componente esté).

---

Para que una persona sea cuidadora de otra deberá tener instalada la aplicación; o sea que al menos deberá ser un usuario pasivo. Se considera usuario activo a aquel que solicita los acompañamientos.

Esta información la consideramos meramente “descriptiva”, sin aporte de alguna posible abstracción, se podría pensar en usar el patrón STATE y que los estados posibles sean “Activo” o “Pasivo” y dependiendo de cuál sea, actuar con su debido comportamiento.
Pero qué comportamiento tendría el estado “Activo”? **“aceptarViaje()**”? “**rechazarViaje()**”? “**recibirNotificación()**”?. Estos métodos consideramos que no son problema de la capa de dominio, sino de la de presentación, son sucesos que le deben “llegar” al usuario cuidador FÍSICAMENTE, al celular si es una aplicación mobile y el tal caso, tienen que ser un botón que dispare un evento desde el FRONT de la app.

---

## Dirección

No tiene comportamiento pero el crear una clase nos permite reutilizarla relacionado a otras clases, por ejemplo el “Viaje”. Además si se quisiera crear un viaje en el que el punto de partida sea la misma dirección de vivienda de la persona transeúnte, la instancia del domicilio de la persona tiene ser la misma que la instancia de la dirección de punto de origen del recorrido, esto no sería posible si el atributo fuese un string.

## Viaje

3. Cada vez que un usuario quiera ir hacia un ***destino***, deberá especificar la dirección exacta donde se encuentra actualmente y la del destino final; además de escoger quiénes serán sus cuidadores (puede haber un solo cuidador). Una vez especificados estos datos, se deberá presionar el botón de *confirmar cuidadores*. Los cuidadores seleccionados por el transeúnte serán notificados y deberán aceptar o rechazar el cuidado.

Decidimos crear una clase viaje que contiene todo lo relacionado a un viaje o recorrido. El transeúnte, sus cuidadores (que mínimo tiene que haber uno), las direcciones de origen y destino, la hora de inicio y la hora estimado de duración, estos dos últimos son datos que servirán para contemplar el hecho de si hubo un accidente o no, que sucede en base al tiempo aproximado que se calculó.

Una comparación del uso de esta abstracción es, por ejemplo, en el desarrollo de un ecommerce, la creación de una entidad “venta” que contiene un carrito con productos y el usuario que realiza la compra, entre otros datos.

“**confirmarViaje()**” es otro problema de la capa de presentación, como “**aceptarCuidado()**” o “**rechazarCuidado()**”

## Distance Matrix API - Tiempo de demora aproximado

4. Si al menos un cuidador acepta la responsabilidad durante el trayecto, al transeúnte se le habilitará el botón de “*comenzar*”. Al ser presionado este botón, el sistema deberá calcular el *tiempo de demora* aproximado y volverle a notificar a sus cuidadores. La distancia (en metros) entre dos direcciones será calculada por “Distance Matrix API” de Google, cuyo sistema nos brinda una interface REST.

Lo único que se sabe de la API es que resuelve la distancia en metros entre dos ubicaciones, pero nosotros necesitamos que se calcule el tiempo aproximado de duración del trayecto.
La opción más simple (no siempre) cada vez que tenemos que usar alguna API es usar un ADAPTER y desarrollar la funcionalidad que nos interesa acoplar a nuestro sistema, por esto, creamos la clase **AdapterCalculoDistanciaViaje,** que resolverá el tema de consumir la API usando el patrón ADAPTER.
Ahora bien, lo que nos ofrece el adapter creado es el poder obtener la distancia en metros entre dos distancias usando la api, pero eso no es el requerimiento pedido. Hay que obtener el tiempo de distancia entre dos direcciones.

Para eso creamos la clase **CalculoTiempoViaje** que tiene un atributo con el adapter para calcular la distancia en metros entre dos direcciones, y el método

**calcularTiempo(Direccion, Dirección)** que es el método que resolverá la lógica del cálculo de tiempo.

---

5. Durante todo el recorrido, el sistema no deberá enviar notificaciones al transeúnte (por motivos de seguridad), ya que el mismo estará en movimiento.

Uno podría estar tentado a usar un patrón STATE en el usuario con los estados **detenido** y **enMovimiento** pero detectar más de un estado no siempre significa usar este patrón. Para usarlo, el comportamiento de la clase debe depender de su estado y en este caso, simplemente dice que no se le deben enviar notificaciones, no aclara que se comporta de una forma u otra. El transeúnte se comporta siempre como un transeúnte, solo que no le llegan notificaciones.

---

6. Una vez que el transeúnte llegue a su destino, deberá presionar el botón "*llegué bien!*". El sistema deberá volver a habilitar las notificaciones, ya que se considera que no hay peligro alguno, y se deberá volver a notificar a sus cuidadores con esta situación.

El "**llegué Bien()**" y "**notificarACuidadores**()" son otros problemas de la capa de presentación.

## Reacciones

7. Si algo malo sucede, el sistema deberá darse cuenta de esta situación por el tiempo aproximado que calculó. El mismo va a reaccionar frente a este incidente según lo que haya configurado el usuario:
   - Enviar un mensaje de alerta a sus cuidadores
   - Realizar una llamada automática a la policía
   - Realizar una llamada al celular del usuario
   - Esperar N minutos para ver si es una falsa alarma (los minutos deben ser parametrizables)

   Se debe considerar que pueden surgir nuevas formas de reaccionar frente a un incidente y que el usuario puede cambiar esta configuración cuantas veces quiera.

"Según lo que haya configurado el usuario" no da pie a pensar que el usuario tiene una forma de reaccionar ante un incidente cada vez que viaja y el último párrafo sugiere que lo puede cambiar cuantas veces quiera. Entonces le agregamos el atributo "**formaDeReacción**" que es del tipo de la interfaz **Reacción** que la implementaran clases por cada tipo de reacción. Esta interfaz tendrá el método reaccionar(Viaje) que recibirá el viaje actual en el que se encuentre el transeúnte y del que obtendrá los datos necesarios para actuar.
Cada clase desarrollará el método dependiendo de su lógica.
La clase **EsperarFalsaAlarma** es un caso particular que tiene una reacción en especial que es la que va a intentar ejecutar en función de los minutos a esperar y podrán darse otras reacciones luego.
La clase **LlamarPolicia** ignora el parámetro viaje.
La clase **LlamarUsuario** necesita el número del usuario, por lo que del viaje utilizará al transeúnte del mismo para obtener este dato.
La clase **EnviarAlerta** necesita a los cuidadores del viaje, los obtendrá del mismo.

## Notificaciones

Ahora bien, las notificaciones hay que modelarlas de alguna forma en la que se puedan enviar notificaciones. Lo que se nos ocurrió fue crear una clase Notificación que contendrá los datos que consideramos necesarios para enviar una notificación, que son el receptor, el mensaje en sí y el emisor, todos son un String, el mensaje tiene sentido que sea así, pero pensando en que las distintas formas de notificar podrían ser varias, no se tendría un tipo de dato concreto. Por eso lo ponemos como String para “incluir” a todas las posibilidades.
Luego, creamos la clase **Notificador** para agregar extensibilidad, que usa esa notificación para enviar la notificación a través del método **notificar(Notificación).** Ahora mismo no sabemos como funciona, pero se nos ocurre que se podrían consumir apis que ofrecen esta funcionalidad y de esta forma en un futuro se puede mejorar la aplicación.

## PUNTO 2

Se decidió realizar un cambio en la clase Viaje, donde el destino ahora es una lista de Destino, una nueva clase en la que se pueden instanciar las direcciones y los minutos por parada por ubicación. Además decidimos que el “método de aviso” sea una interfaz y el tiempo se calcule ahora mediante un método que utiliza el “método de aviso” para realizar el cálculo de tiempos del viaje calculado con polimorfismo según el “método de aviso”. La clase **CalculoTiempoViaje** se sigue utilizando de la misma manera que antes pero ahora dentro de cada uno de los elementos de la interfaz “método de aviso”. De esta manera también si en un futuro se desean agregar “métodos de aviso” sería más sencillo.