Punto 1

Diseño Inicial

Inicialmente pensamos el diseño con un patrón State, pero luego nos dimos cuenta que no tenía sentido debido a que no existe comportamiento común entre los estados “Transeúnte” y “Cuidador”. La aplicación de Uber, por ejemplo, permite a los usuarios monitorear los viajes de sus contactos. Entonces, no hay diferenciación entre un “Transeúnte” y un “Cuidador”, sino que un transeúnte es un usuario pidiendo un viaje, y un cuidador puede ser cualquier usuario que tenga descargada la aplicación. Finalmente, llegamos a la conclusión de diagramar una sola clase llamada Persona para simplificar la idea mencionada anteriormente.

Comportamiento

El comportamiento sería el siguiente: una persona solicita a varios cuidadores, un **“pedidoDeCuidado”** para un viaje de un destino a otro. Los cuidadores (personas) pueden aceptar el cuidado o no recibiendo quién se lo solicitó y el viaje en cuestión. En el momento que al menos una persona acepte el cuidado, se creará un viaje. El viaje deberá ser capaz, mediante la herramienta “Distance Matrix API” (que nos devuelve la distancia entre ambos puntos), calcular un aproximado de su tiempo de demora. El sistema es el encargado de monitorear la duración del viaje y de constatar que coincida aproximadamente con el estimado calculado previamente. En caso de que esto no suceda, se deberá emplear la reacción elegida por el usuario. La notificación que recibe el usuario, el botón “comenzar” para el inicio del viaje y el botón de “llegué bien”, le corresponden a otras instancias de diseño, por lo que no serán parte del modelado de esta capa.

Una vez terminado el viaje, se deberá notificar a sus cuidadores de la finalización del mismo.

Patrón Strategy e implementación

Identificamos el comportamiento del patrón Strategy entre la clase Persona y las reacciones que puede ejecutar. Una persona debe ser capaz de cambiar de reacción como desee desde su configuración, dentro de las distintas reacciones disponibles. En caso de crear una nueva reacción, se crea una nueva estrategia polimórfica, sin necesidad de impactar el resto de la lógica. Además, agregamos la posibilidad de esperar N minutos a cualquier reacción. Esta misma no la implementamos como otro tipo de reacción, sino como si fuera un opcional previo a las mismas, ya que cualquier reacción debería poder esperar N minutos antes de ejecutarse (o al menos así lo entendimos).

La implementación de la función que ejecuta las reacciones quedaría de esta manera:

```
void emplearReaccion() {
    if (esperarNMinutos) {
        realizarEspera.esperarYLuegoEjecutar(reaccionElegida);
    } else {
        reaccionElegida.ejecutarReaccion();
    }
}
```

Patrón Adapter

Pensamos en aplicar el patrón Adapter para poder obtener la distancia en metros entre dos Ubicaciones mediante el Distance Matrix API, y luego así calcular la demora aproximada.

Punto 2

Aclaración

Primero que nada, nos parece adecuado aclarar que si bien el enunciado puede dar a entender que las demoras son el tiempo parado entre trayecto y trayecto, basándonos en la forma en que está escrita la consigna - la idea del “cuidado sobre el trayecto”- y aclaraciones por parte de los ayudantes, decidimos que esta funcionalidad calculara los tiempos aproximados de cada trayecto de un viaje con múltiples paradas.

Modificaciones e Implementación

Para cumplir con el requerimiento realizamos una serie de modificaciones pensando en las nuevas condiciones planteadas. La clase Viaje debe contar con una lista de ubicaciones, del tipo Ubicación. Luego, dentro de Viaje vamos a calcular la distancia **(calcularDistancia(Ubicacion origen, Ubicacion final))** y con esto obtener la duración del trayecto **(duracionDelTrayecto(Integer distancia))** entre una ubicación y otra. Finalmente, calculamos el tiempo final aproximado de demora **(calcularTiempoTotalAprox())**. La parte del diagrama que involucra a la API no se vé afectada porque se encarga únicamente de calcular la distancia entre dos ubicaciones.

La implementación de **calcularTiempoTotalAprox()** quedaría de la siguiente manera:

```
int calcularTiempoTotalAprox() {
    tiempoTotal = 0;
    for (i = 0; i <= (viaje.size() - 1); i++) {
        int distancia = calcularDistancia(viaje[i], viaje[i + 1]);
        tiempoTotal += calcularDuracionTrayecto(distancia);
    }
    return tiempoTotal;
}
```