Diseño de clases

Trayecto

Usamos esta clase para los trayectos recorridos con nuestro sistema. Principalmente un trayecto tiene un transeúnte, origen, destino y cuidadores.

Los puntos de Origen y Destino son representados con instancias de Dirección.

El transeúnte es la persona que solicitó el trayecto, sólo nos interesa saber la instancia de Persona pues con eso ya tenemos acceso a los métodos que necesitamos.

Los cuidadores se representan con una lista de Personas, compuesta por cada Persona que aceptó la responsabilidad del cuidado.

Dirección

Representa una dirección urbana en alguna ciudad o pueblo de Argentina.
Necesariamente nos interesa saber Calle, Altura, Código Postal, Localidad, Partido y Provincia.

Persona

Representa a una persona que usa la aplicación. Registra su Nombre, Apellido, Sexo y Edad. También guarda su dirección como una instancia de Dirección, su usuario como instancia de Usuario y su inbox de notificaciones como una instancia de Inbox.

Usuario

Representa a un usuario de la aplicación. El propósito de esta clase es encapsular los datos y funcionalidades que hacen a la experiencia del usuario con la aplicación. De momento solo registra el nombre de usuario y su tipo.

Tipo de Usuario

Puede ser Activo o Pasivo. Un usuario Activo puede solicitar cuidados (Transeúnte) mientras que un usuario Pasivo puede ser requerido para un cuidado.

Notificación

Representa un mensaje que el sistema le envía a una persona y que además se encarga de notificarla.

El mensaje se representa con una String, y se dispone de un Booleano para indicar si la notificación fue enviada o no. El sistema asume que toda notificación enviada fue leída.

Inbox

Representa al inbox de notificaciones de una Persona, similar a un “buzón de notificaciones”.
Cada instancia de esta clase registra todas las notificaciones que se le enviaron a una persona como elementos de una colección de Notificación.

También tiene un Booleano “silenciado” que indica si el Inbox está silenciado o no.

Reacción Incidente

Utilizamos el **patrón Strategy**, implementando la posibilidad *“Estrategia como interfaz”*. Nuestra *Estrategia* en este caso será la interfaz IReaccionIncidente, nuestro *Contexto* será Trayecto y nuestra función *ejecutar()* será reaccionar(Trayecto).

Cada implementación de IReaccionIncidente será un tipo de reacción, que deberá resolver cómo reaccionar ante un incidente para el trayecto dado.

Inicialmente partimos con las reacciones Alertar Cuidadores, Llamar Policía, Llamar Transeúnte y Prorroga.

Alertar Cuidadores

Enviar una notificación a los cuidadores del trayecto con el siguiente mensaje:

> *“¡Atención, {{Nombre del transeúnte}} tuvo un incidente en el trayecto!”*

Llamar Policía

Se realiza un llamado al 911 registrado como constante en la misma clase.

Se utiliza el teléfono del sistema para realizar el llamado, al mismo se le envía el siguiente mensaje:

> *“{{Nombre del transeúnte}} tuvo incidente mientras se dirigía desde {{Origen del trayecto}} hacia {{Destino del trayecto}}”*

Llamar Persona

Se realiza un llamado a un número de teléfono proporcionado anteriormente, por defecto al del transeúnte. El llamado será mudo y es sólo para avisarle a la persona que el sistema asumió un incidente.

Prorroga

Se espera una determinada cantidad de minutos en caso de que sea una falsa alarma. Los minutos fueron configurados por el Transeúnte al momento de registrar la reacción.

Llamador Telefónico

Representa al “teléfono” del sistema se ocupa de realizar llamados telefónicos de parte del sistema. Provee la función “*llamar(teléfono, mensaje)*” que llama al número de teléfono proporcionado y, en caso de que alguien atienda lee en voz alta el mensaje dado por una String.

El propósito de esta clase es abstraer la lógica de cómo realizar un llamado telefónico para que el sistema pueda reutilizarla cada vez que necesite hacerlo. En principio sólo será con las reacciones Llamar Policía y Llamar Persona.

Estimador Demoras

Utilizamos el **patrón Adapter**. Siguiendo la estructura genérica del patrón, tenemos la interfaz "IEstimadorDemoras" que representa al adaptador y se encarga de estimar la demora entre dos direcciones y devolver su valor en metros. El Trayecto solo necesita conocer la demora para asumir un incidente o no.

Implementamos la clase "DefaultEstimadorDemoras" que siguiendo la estructura genérica del patrón Adapter sería nuestro "adaptador concreto" para la clase "DistanceMatrixAPI" que sería nuestra "adaptada" al api de Google.

El "DefaultEstimadorDemoras" o Estimador por defecto usa la api de Google para saber la distancia entre dos direcciones y luego la multiplica por la constante de velocidad de peatones para obtener una estimación en minutos de cuánto tardará el transeúnte en recorrer el trayecto.

Precondiciones

1. Todos los llamados telefónicos se hacen desde el servidor.
2. El número de la policía es constante y es 911, si se requiere llamar a otro número habrá que usar Llamar Persona o implementar una reacción distinta.
3. Todas las distancias están dadas en metros y las demoras en minutos. Redondeadas a números enteros ya que no nos interesa trabajar con fracciones de minutos ni de metros.
4. Se asume como constante de velocidad promedio de cualquier Transeúnte 3,5 km por hora = 50 metros por minuto (aproximadamente 1 cuadra cada 2 minutos). Dado que, basándonos en esta fuente, la misma oscila entre 3 y 4,5 km por hora.

Patrones descartados

- Consideramos el uso del **Patrón State** para modelar los estados del Trayecto como "solicitado", "confirmado", "en curso" y "finalizado". Lo descartamos porque nos resultaba un diseño un poco más complejo y porque queríamos que Trayecto entienda los mensajes Comenzar y Finalizar en lugar de algo del estilo *estado.Actualizar(Trayecto)*.
- Consideramos el uso del **Patrón Adapter** para el teléfono del sistema. Lo descartamos porque decidimos no mantener múltiples estrategias para efectuar llamados telefónicos.

Modificaciones punto 2

Modificamos Trayecto para que en lugar de Origen y Destino tenga una lista de secciones, además agregamos el concepto de Estrategia de Trayecto para resolver cómo asumir incidentes y calcular demoras según si el usuario decidió ir avisando punto a punto o que el sistema asuma una demora de N minutos por parada. Cada trayecto instancia su propia estrategia.

Sección

Una sección es sólo un tramo del recorrido que representa Trayecto. Principalmente tiene Origen y Destino indicando de dónde y hacia dónde es este recorrido, y una referencia al trayecto al que pertenece. El propio Trayecto es responsable de guardar el orden de las secciones, para esto elegimos usar una Lista de secciones donde todas las secciones ya estarán ordenadas en el Trayecto.

Estrategia

Ahora los trayectos definen su estrategia de trayecto. Una estrategia es una clase que define cómo calcular la demora de una sección, de un trayecto, y también cómo/cuando asumir un incidente en un trayecto.

Para calcular demoras usamos algo parecido a **Template Method**, ya que la demora del trayecto se calcula siempre como una sumatoria de las demoras de cada sección. Nuestro método template es "calcularDemora(Trayecto)" que siempre hace la misma suma, pero cambia la forma de calcular los términos de esta suma. El "paso" sería "calcularDemora(Sección)" y cada estrategia implementa su cálculo de demora concreto.

Utilizamos el **patrón Strategy** para asumir incidente, nuestra estrategia base no define ninguna forma de cómo resolver esto y queda completamente atado a cada implementación cómo y/o cuando asumir un incidente.

Estrategia Por Paradas

Se guarda el parámetro "demoraPorParadas" indicando los N minutos que el usuario se detendrá en cada parada.

Para el cálculo de demora de una Sección, se usa el estimador de demoras para el Origen y Destino de esa sección y al resultado se le suma la demora por paradas configurada.

Se asume un incidente si el usuario no llegó al final del trayecto antes de la demora total del trayecto, calculada según el método base de la clase abstracta "EstrategiaTrayecto".

Estrategia Punto a Punto

Se calcula la demora de una sección solo con el estimador de demoras para el Origen y Destino de la sección.

Se asume un incidente si luego de esperar los minutos de la demora de la sección, el usuario aún no avisó que llegó al destino de esta.

Avanzar()

Agregamos este método al Trayecto para representar la acción de que el usuario llegó al destino de la sección que estaba recorriendo. Dependerá de cada estrategia ver si esta información se usa o no.

Pseudocódigos

```
public abstract class EstrategiaTrayecto {
    public Trayecto trayecto;
    new *
    public abstract void asumirIncidente();
    new *
    public abstract int calcularDemora(Seccion seccion);
    new *
    public int calcularDemora(Trayecto trayecto) {
        return trayecto.Secciones.Sum(s => calcularDemora(s));
    }
}
```

```
public class EstrategiaPorParadas inherits EstrategiaTrayecto {
    new *
    public int calcularDemora(Seccion seccion){
        return estimadorDemoras.estimarDemora(seccion.Origen, seccion.Destino) + demoraPorParadas;
    }
}

new *
public class EstrategiaPuntoAPunto inherits EstrategiaTrayecto {
    new *
    public int calcularDemora(Seccion seccion){
        return estimadorDemoras.estimarDemora(seccion.Origen, seccion.Destino);
    }
}
```