Punto 1

Se modeló una única clase concreta Persona en vez de una clase concreta Transeúnte y otra Cuidador, dado que un transeúnte puede actuar como cuidador y un cuidador puede actuar como transeúnte.

Se modeló Destino como clase concreta debido a que la queremos tratar como una instancia desechable. Es decir, en caso de querer cambiar algún atributo, simplemente la desechamos e instanciamos una nueva. Esto es posible dado que las clases concretas son extensibles en tiempo de ejecución.

Género podría haber sido modelado como un enumerado pero priorizamos la comodidad del usuario para quienes no se sienten incluidos al optar por la opción “Otros” entre “Masculino” y “Femenino”. Lo modelamos como clase concreta porque le damos mayor libertad de elección al usuario al ser extensible en tiempo de ejecución y porque sabemos que el género no es un aspecto relevante para nuestro sistema.

Se decidió no modelar la diferenciación entre usuario activo y pasivo por KISS y YAGNI dado que el único aspecto relevante de esto por el momento es que, para que una persona sea cuidadora de otra deberá por lo menos tener instalada la aplicación, y esto se cumple para todo usuario.

Se decidió modelar la sección del notificador utilizando el patrón Strategy dado que este nos permite:

- Cambiar de TipoDeNotificacion en tiempo de ejecución.
- Es extensible en tiempo de ejecución por lo que nos permite agregar nuevos TiposDeNotificacion en tiempo de ejecución.
- Nuestras estrategias, que son las clases NotificarSolicitudDeCuidado, NotificarComienzoDeCuidado y NotificarFinDeCuidado, quedan más testeables, analizables, modulares y cohesivas.
- La clase Notificador tiene menor acoplamiento y mayor cohesión que si se hubieran modelado las diferentes estrategias como métodos de esta clase.

Se modeló CalculadoraDeTiempoDeDemora como clase concreta para mantener nuestras clases altamente cohesivas y con bajo acoplamiento. Esto también es válido para la clase Viaje.

Para el cálculo de la distancia, decidimos utilizar el patrón Adapter para bajar el acoplamiento con la “Distance Matrix API”. De esta manera, si en un futuro la API quedase deprecada, sería fácilmente reemplazable por otra clase que implemente la interface AdapterDistancia. Además, el patrón nos brinda:

- Mayor mantenibilidad debido al bajo acoplamiento entre la clase CalculadoraDeDistancia y la clase DistanceMatrixAPI.
- Mayor cohesión de la clase CalculadoraDeDistancia debido a la delegación de cierto comportamiento en el adapter.
- Mayor facilidad de testeo ya que se podría implementar un mock de AdapterDistanceMatrixAPI.

Con respecto a que el sistema no deberá enviar notificaciones al transeúnte cuando está en movimiento, decidimos modelar un atributo booleano notificacionesHabilitadas en la clase Persona. De esta manera, el notificador enviará o no el mensaje dependiendo del valor de este atributo.

Se decidió modelar la sección del manejo de incidentes utilizando el patrón Strategy dado que este nos permite:

- Cambiar de RespuestaAIncidente en tiempo de ejecución.
- Es extensible en tiempo de ejecución por lo que nos permite agregar nuevas RespuestasAIncidentes en tiempo de ejecución.
- Nuestras estrategias, que son las clases AlertarCuidadores, LlamarALaPolicia, LlamarAlCelularDelUsuario y EsperarPorSiEsUnaFalsaAlarma quedan más testeables, analizables, modulares y cohesivas.
- La clase ManejadorDeIncidente tiene menor acoplamiento y mayor cohesión que si se hubieran modelado las diferentes estrategias como métodos de esta clase.

Se modelaron los atributos celular y cantidadDeMinutosPorFalsaAlarma en la clase Persona para lograr un menor acoplamiento y una mayor cohesión, dado que son utilizados por LlamarAlCelularDelUsuario y EsperarPorSiEsUnaFalsaAlarma respectivamente.

La clase EsperarPorSiEsUnaFalsaAlarma es un CronTask que ejecuta durante los N minutos configurados por el usuario.

No se graficaron varias relaciones de uso para brindar mayor claridad en el diagrama de clases.

Punto 2

Se le agregó una lista de Destinos a la clase Viaje para que los transeúntes puedan escoger destinos con varias paradas.

Decidimos crear un único método "calcularTiempoDeDemoraAproximadoEnMinutos(List<Destino> paradas): Integer", el cual se encuentra en "src/main/java/ar/utn/frba/dds/CalculadoraDeTiempoDeDemora.java" dado que el usuario posee dos opciones configurables no excluyentes: especificar si se detendrá N minutos en cada parada o ir avisando punto a punto su estado de salud.

El atributo cantidadMinutosADetenerse de la clase Destino será seteado como -1 en caso de optar por ir avisando punto a punto su estado de salud en esa parada.