Implementación en pseudocódigo del cálculo de la demora

```
import java.time.Duration;

class Seccion{
	Ubicacion origen, destino;
	Duration demora = 0;		//Se solicitará la demora en el generador de Seccion
	APIAdapterCalculadorDistancia calculadorDistancia;

	Duration calcularDuracion(){
		return this.calcularTiempoEstimadoDeSeccion() + this.demora;
	}

	Duration calcularTiempoEstimadoDeSeccion(){
		double distanciaARecorrer =  this.calculadorDistancia.calcularDistancia()
		return distanciaARecorrer/velocidadPromedio
	}
}


class Recorrido{
	List<Seccion> secciones;
	Duration tiempoEstimado;

	Void calcularTiempoEstimado(){
		this.tiempoEstimado = this.secciones.stream()
		.forEach(s -> s.duracionSeccion()).sum();
	}
}
```

Uso de patrón Strategy como solución a Reacción al incidente

Contexto:

Se indica que pueden surgir nuevas formas de reaccionar frente a un incidente y que esta puede cambiar según decisión del usuario.

Solución:

Utilizando el patrón Strategy nos permite generar instancias de las subclases de reacción que implementan cada una el método reaccionar de forma distinta a las otras subclases. A la vez permite el cambio de reacción a partir de la creación de nuevas instancias de las distintas subclases. Esta solución brinda cohesión a las clases usuario y recorrido las cuales no se encargan de solucionar la implementación de la reacción ante un incidente si no que esto se delega a cada una de las subclases. Además brinda mantenibilidad al permitir verificar cada implementación del método reaccionar independientemente del resto.

Uso del patrón adapter como solución a la implementación de la API

Contexto:

El cálculo del tiempo de demora del recorrido depende de la distancia entre el origen y el destino el cual se obtiene a partir del uso de una API de cálculo de distancia.

Solución:

Utilizando el patrón adapter nos desacoplamos de la implementación de la solución del cálculo de la distancia. De este modo el adaptador se encarga de acomodar las diferencias de la implementación del lado de la API a las funcionalidades definidas en nuestro modelado.

Aclaraciones

1. El atributo ‘activo’ de la clase ‘usuario’ restringe la posibilidad del usuario para pedir un recorrido, aceptar recorrido y otras acciones relacionadas a los recorridos. Solamente puede comenzar y, luego, terminar el recorrido (dándolo por finalizado), y ahí se vuelve a su estado original (inactivo). Este atributo se setea en ‘True’ cuando se genera un recorrido o cuando se acepta ser cuidador de uno.
2. El método generarRecorrido() genera tanto el recorrido como las notificaciones ‘cuidadorSolicitado’ las cuales avisan a los cuidadores disponibles del recorrido, para que estos puedan o no aceptarlo
3. La clase Notificador se encarga del envío de notificaciones. Esta genera notificaciones dependiendo del método que se llame en la clase Recorrido
4. Calle como clase para contemplar consistencia de datos
5. Sexo como clase para contemplar consistencia de datos
6. Clase Mensaje con horaEnvio para trazabilidad de datos