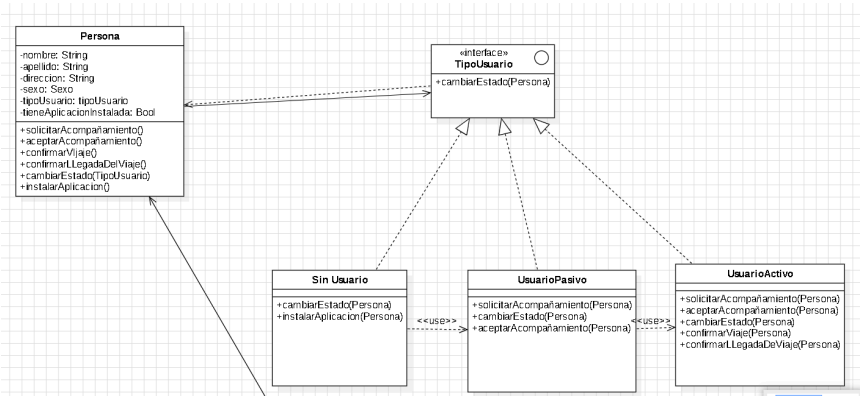
Diseño de sistemas

Justificaciones de Diseño

Cuidándonos: Práctica Entregable

- Curso: K3152
- Alumno: Gabriel Borré

Clase Persona y clase TipoUsuario



Para modelar la clase Persona, utilicé un **Patrón State**, puesto que el comportamiento que una instancia de dicha clase puede tener dependen del estado de la misma. Por eso mismo, además de los atributos básicos que esta clase tendrá (nombre, apellido, dirección y sexo), decidí oportuno incluir un atributo que sea una instancia de la clase TipoUsuario, dentro de la clase Persona. De esta manera, cuando se le envíe un mensaje a la persona relacionado con una de las acciones que puede realizar (por ejemplo, solicitarAcompañamiento), lo que hará ese método dentro de la clase persona será enviarle un mensaje a la instancia del tipo de Usuario que le corresponda, de modo que la clase a la cual ella pertenece sea la encargada de definir la implementación del método puesto como ejemplo.

Por ejemplo:

Clase Persona

```
public class Persona{

        private tipoUsuario: TipoUsuario;

        public void solicitar Acompañamiento( ){
```

```
            tipoUsuario.solicitarAcompañamiento(this )

        }



}
```

De esta forma, la instancia de la clase persona delega en la instancia de la clase TipoUsuario la implementación del método solicitarAcompañamiento ( ), lo cual le brinda al sistema mayor cohesión. Al mismo tiempo, cuando una persona solicita un acompañamiento, su estado interno cambia, ya que deja de ser un usuario pasivo para convertirse en un usuario activo. Este cambio de estado también es implementado dentro de la clase UsuarioPasivo, en el cuerpo del método solicitarAcompañamiento.

Clase Usuario Pasivo

```
public class UsuarioPasivo implements TipoUsuario{

        public void cambiarEstado(Persona persona){

                persona.cambiarEstado(new UsuarioActivo())

        }
        public void solicitarAcompañamiento(Persona persona ){
                this.cambiarEstado(persona)
                // implementación del metodo


        }


}
```

El uso de este método también es necesario dado que la persona puede acceder a ciertas funcionalidades, dependiendo de su estado. Por ejemplo, una persona registrada en el sistema, pero que no tenga la aplicación instalada,no debería poder solicitar un acompañamiento como transeunte, ni tampoco debería poder aceptar un acompañamiento en condición de cuidador. Tampoco un usuario pasivo debería acceder al método aceptarViaje, ya que para esto, primero debería haber solicitado un acompañamiento, lo cual lo transforma en un usuario activo. En virtud de lo primero, decidí crear una clase SinUsuario, la cual convierta a una persona que tiene como atributo a una instancia de SinUsuario, a una persona que contenga como atributo una instancia de la clase UsuarioPasivo, para que, de esta manera, pueda acceder a los métodos propios de la clase.

Por último, es importante destacar que todas las clases implementan la interfaz TipoUsuario, la cual contiene la firma del método cambiarUsuario(Persona), dado que las clases SinUsuario y UsuarioPasivo deben poder modificarle el tipo de usuario a la persona.

Puesto en pseudoCodigo:

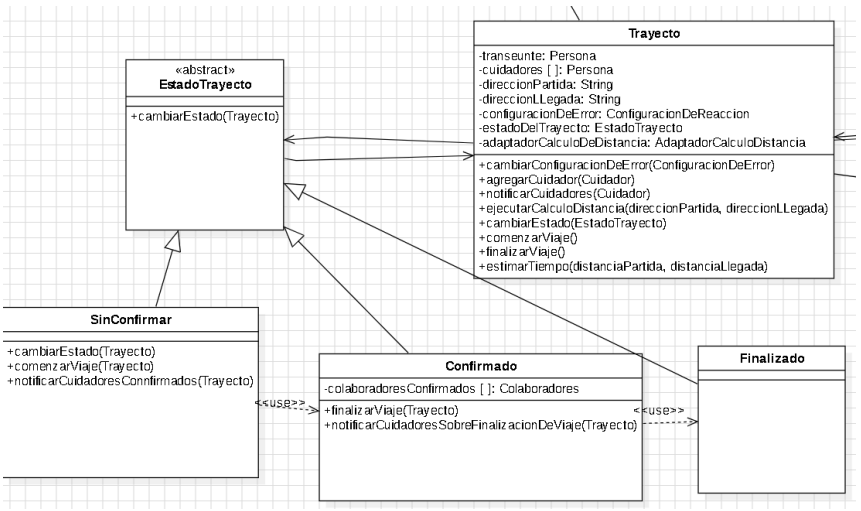
Clase Persona

```
public class Persona{


        method instalarAplicacion{
                tipoUsuario.instalarAplicacion(this)

        }

}
```

Clase Sin Usuario

```
public class SinUsuario implements TipoUsuario{

public void cambiarEstado(Persona persona){
        persona.cambiarEstado(new UsuarioPasivo())


}

public void instalarAplicacion(Persona persona){
        persona.cambiarEstado(persona)
}


}
```

Clase Trayecto y clase EstadoTrayecto



La clase Trayecto se creó con en virtud del objetivo principal de la aplicación: que las personas puedan viajar de una dirección de partida a una dirección de destino, teniendo en cuenta que esta persona debe ser tutelada por un grupo de cuidadores.
En consecuencia, la clase tendrá como atributos un transeunte (instancia de la clase Persona), un grupo de cuidadores (lista con instancias de la clase Persona), una dirección de Partida y una dirección de llegada. Asimismo, el trayecto tiene diferente comportamiento en función del estado que presente. Por ejemplo, un trayecto que se encuentra sin confirmar a causa de que el transeunte lo ha solicitado pero sus cuidadores no lo han confirmado, no puede acceder al método finalizarViaje( ). Debido a que los mensajes que se le pueden enviar a una instancia de la clase Trayecto dependen del estado, decidí implementar nuevamente un **Patrón State**, por los mismos motivos dados en la clase Persona: su comportamiento (el cual abarca los métodos a los que puede acceder) depende de su estado y, además, determinados métodos son los encargados de modificar el estado interno del trayecto. Por ejemplo, cuando se comienza un viaje porque un cuidador aceptó acompañar al transeunte, el trayecto (que inicialmente se encuentra sin confirmar) pasa al estado confirmado y quien se encarga de modificar el estado del trayecto es la instancia del estado SinConfirmar

Implementación en pseudocódigo

Clase Trayecto

```
public class Trayecto {


        private estadoDelTrayecto : EstadoTrayecto

        public void cambiarEstado (estadoTrayecto EstadoTrayecto){

                this.estadoDelTrayecto=estadoTrayecto

}
```

```
	public void comenzarViaje(){

	estadoDelTrayecto.comenzarViaje(this)

}


}
```

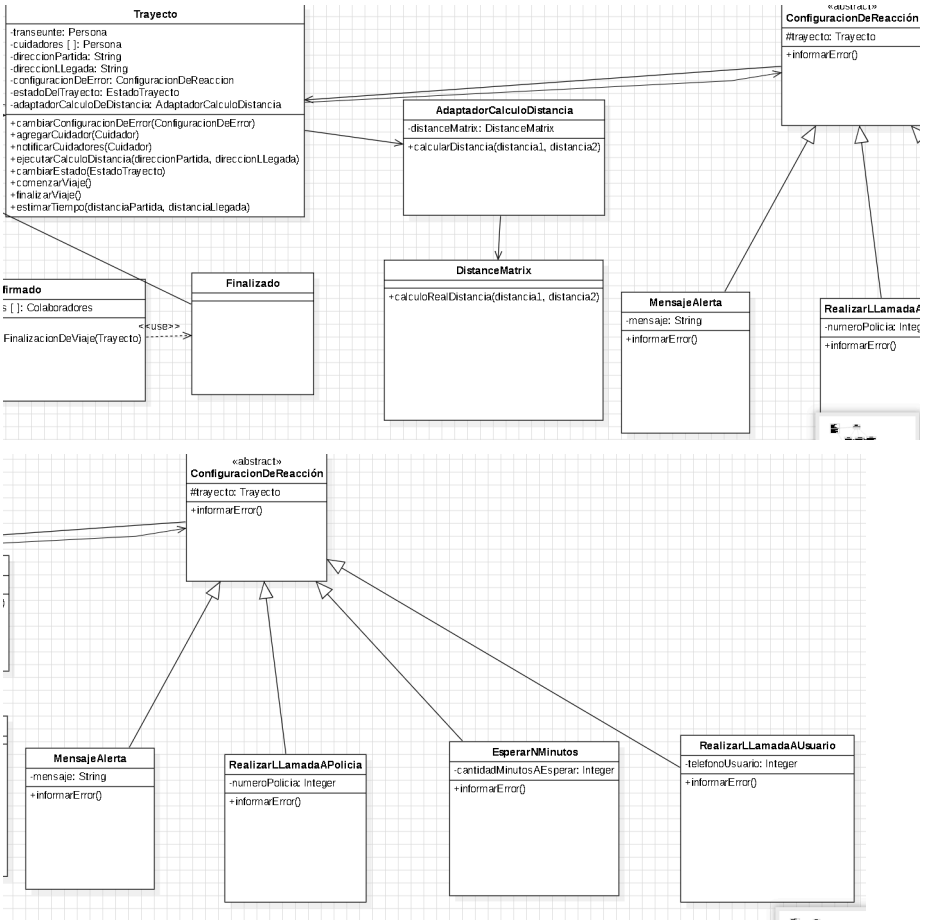
Clase Sin Confirmar

```
public class SinConfirmar implements EstadoTrayecto{

public void comenzarViaje( trayecto Trayecto){
		private Persona cuidadoresConfirmados [ ]
		if(trayecto.cuidadores().stream().anyMatch(colaborador ->
colaborador.confirmarViaje()){


cuidadoresConfirmados=trayecto.cuidadores().stream().filter(cuidador->cuidador.confirmarVi
aje())
				trayecto.cambiarEstado(new Confirmado(cuidadoresConfirmados))


		}



}



}
```

Lo mismo ocurre con las otras clases. Al método finalizarViaje solamente puede acceder la clase Confirmado, lo que quiere decir que un trayecto se puede finalizar si y solo si se encuentra confirmado. Este método implementaría una lógica similar al anterior: la instancia de la clase trayecto llama en el método finalizarViaje a una instancia de la clase Confirmado, la cual tiene el método implementado. Dicho método va a modificar el estado del viaje y lo pondrá en finalizado.

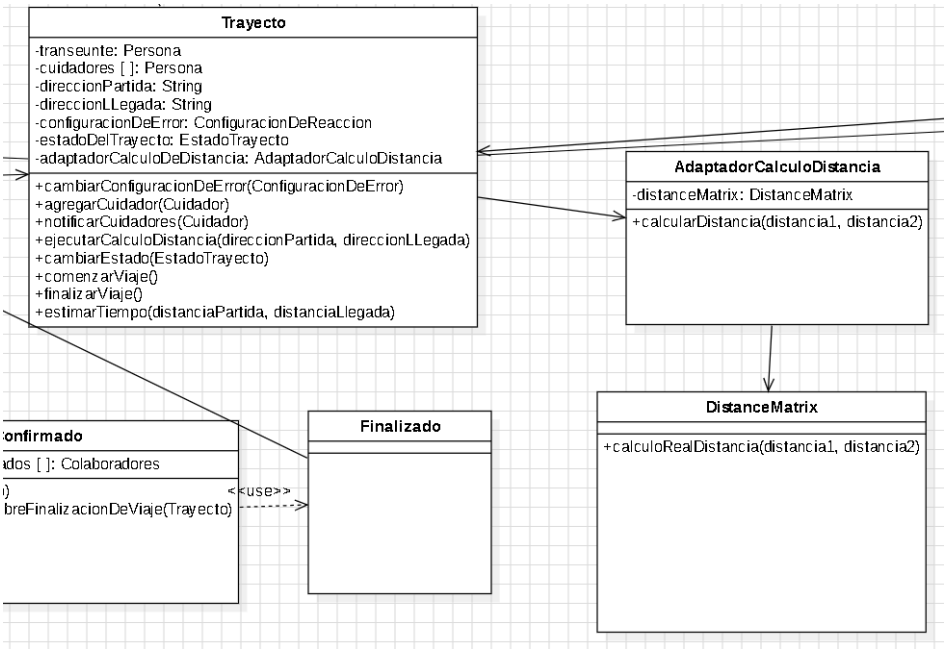
Por otro lado, es importante conocer el estado del trayecto ya que una de las reglas del negocio enuncia que los transeúntes que se encuentren en viaje no deben recibir notificaciones. Nuestro modelo nos permite tener en cuenta esto, dado que por cada trayecto que se encuentre en el estado Confirmado, sabemos que el transeunte no puede recibir mensajes.

Clase Trayecto y clase ConfiguraciónDeReacción



La clase trayecto, además de incluir los atributos mencionados anteriormente, debe contener un atributo relacionado a la acción que debe realizar el sistema cuando el viaje del transeúnte demora más de lo estimado. Dicha acción responde a una configuración de reacción estipulada por el transeúnte, una vez que decide solicitar un acompañamiento para un trayecto. A causa de que existen diferentes configuraciones de reacción y que, además, se podrían ir agregando nuevas configuraciones a lo largo del tiempo, es conveniente utilizar el **Patrón Strategy** . Por un lado, la clase Trayecto va a contener una instancia de la clase ConfiguraciónDeReacción. Por otro lado, cada forma de configurar un error va a ser una clase que implemente su manera el método informarError(), el cual heredarán de la clase abstracta ConfiguraciónReacción. Cada una de estas subclases, implementará a su manera dicho método.
Utilizar el patrón Strategy nos proporciona mayor **cohesión** para la clase Trayecto, ya que esta clase no será la encargada de implementar un método por cada posible configuración de reacción, sino que eso será responsabilidad de cada subclase de la clase abstracta ConfiguraciónReacción. Asimismo, este método nos provee **flexibilidad**, dado que si en el futuro se quisieran agregar nuevas formas de reaccionar, solamente deberíamos crear una clase nueva que herede de la clase abstracta ConfiguraciónDeReacción y que implemente a su manera el método informarError().

Clase Trayecto y Clase AdaptadorCalculoDistancia



Debido a que la clase Trayecto delega en “Distance Matrix API” de Google el cálculo de la distancia entre la dirección de partida y la dirección llegada de un trayecto, se implementó el **patrón Adapter** , debido a que queremos seguir adelante con la implementación de un método que calcule la distancia entre dos direcciones sin saber cómo se hará, puesto que la implementación no está definida por nosotros, sino por Google, en este caso. Por este motivo, la clase Trayecto contendrá como atributo a una instancia de la clase AdaptadorCalculoDistancia, la cual a su vez incluirá una instancia de la clase DistanceMatrix, el cual implementará en su método calculoRealDistancia (distancia1, distancia2) la manera en la que Google calcula la distancia entre dos direcciones.

Implementación en PseudoCódigo

Clase Trayecto

```
public class Trayecto{

        public AdaptadorCalculoDistancia adaptadorCalculoDeDistancia;
        public String direccionPartida;
        public String direccionLlegada;

        public void ejecutarCalculoDistancia(String direccionPartida,String direccion llegada){

        adaptadorCalculoDeDistancia.calcularDistancia(direccionPartida,direccionLLegada)
        }


}
```

Clase AdaptadorCalculoDistancia

```
public class AdaptadorCalculoDistancia{
        public DistanceMatriz distanceMatrix;

        public void calcularDistancai(String direccionPartida, String direccionLLegada){

                distanceMatrix.calculoRealDistancia(direccionPartida,direccionLLegada)

        }



}
```

Con este patrón, aumentamos la cohesión en la clase Trayecto, ya que esta no es responsable de implementar un método que calcule la distancia entre dos direcciones.