Proyecto MoviFast

Desarrollado por: Christian Collaguazo

Programación 3.

Universidad de Cuenca.

Semestre: Marzo – Agosto

Cuenca - Ecuador – 2017

Contenido

1. Introducción

Hoy en día la movilidad en Cuenca se encuentra en un gran avance con proyectos que prometen dar a los ciudadanos varias opciones al momento de transportarse a distintos puntos en la cuidad.

MoviFast plantea ser una herramienta para facilitar dicha movilidad, que tiene como objetivo el sector del trasporte terrestre en modalidad Taxi(convencional), el cual ayudaría a los usuarios de dicho servicio tener una idea de la ruta en la cual efectuaran uno o varios viajes en unidades de taxi, dando herramientas las cuales permitan dar a conocer costes, rutas optimas que ayuden al usuario a llegar a su destino ahorrando tiempo y dinero.

2. Descripción del Proyecto

El proyecto se centra en la mejora del uso del sistema de transporte publico terrestre, ya que hoy en día los usuarios nos hemos limitado a solo saber la información de una carrera al momento de tomar un taxi y llegar a nuestro destino, la aplicación sería una herramienta que daría a los usuarios el control de poder elegir la ruta más óptima para poder llegar a su destino permitiendo saber además el coste y tiempo aproximado del viaje.

Así mismo el programa contará con una interfaz la cual permitirá a los usuarios poder ingresar dichos destino o destinos a viajar. En la Fig 1 se muestra como es dicha interfaz.

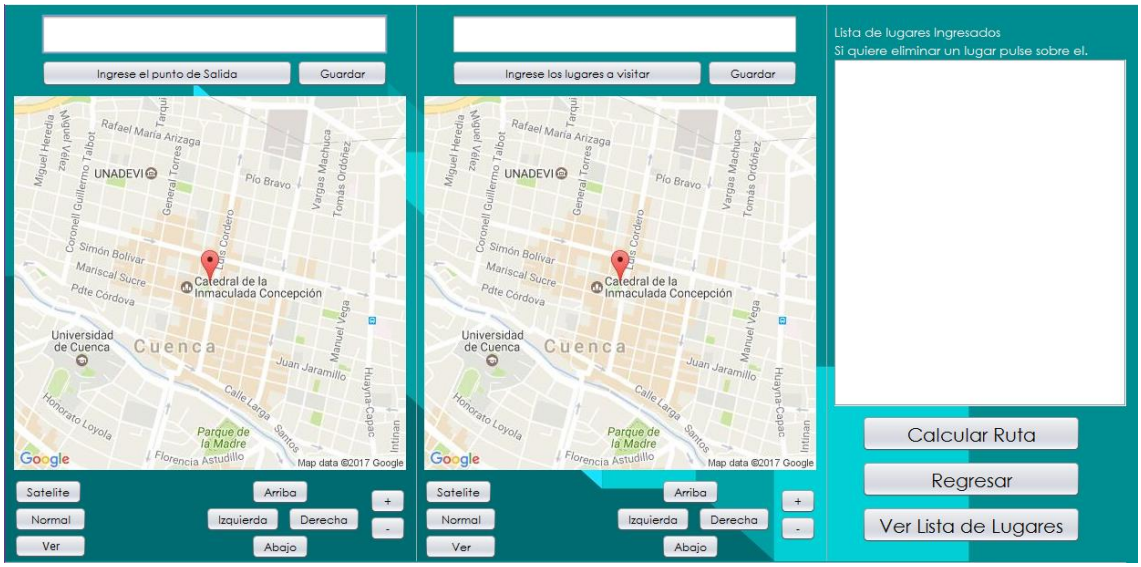

Fig. 1: Interfaz de usuario para ingreso de rutas

A demás el programa cuenta con la capacidad de obtener las rutas mínimas para poder viajar a distintos puntos que el usuario ingrese al sistema dicho capacidad se da gracias a la ayuda de un árbol de expansión mínima el cual fue modificado su construcción para permitir viajar a todos los puntos solo pasando por ellos una solo ves más adelante se hablara de cómo se consiguió esto.

Un estudio hecho sobre las tarifas de taxi en Cuenca permite a la aplicación aproximar el costo de dicha carrera teniendo en cuenta la distancia de la ruta más corta para llegar a un destino, por su parte otros factores que influyan sobre estos valores no han sido analizados esta vez los cuales tomarían un poco más de tiempo analizar, pero la proyección hacia el futuro es que dicho valor que salga en la aplicación sea el valor que se cobre al usuario al finalizar la carrera.

El proyecto cuenta con la capacidad con la cual en un futuro se planea ponerlo en uso ya dentro del contexto real en la ciudad, mientras tanto esto se simulara para dar una idea de cuáles son los planes en el futuro, la idea es tener una interfaz la cual nos mostrara diferentes unidades de taxi las cuales se nos mostraran en un mapa de la ciudad las cuales estarán disponibles para el uso del usuario, el cual permitirá llamar a la línea de taxi más cercana para poder hacer uso de ella, a la cual se le mandara la información del usuario y la información de la ruta que se quiere visitar.

Cabe decir que toda información de las rutas se obtiene gracias a las diferentes opciones que nos brinda el api de Google Maps la cual se hablara de su uso más detalladamente en otra sección del informe.

3. Objetivos del Proyecto

A lo largo del pensamiento del proyecto se plantearon objetivos los cuales fueron.

- Diseñar una aplicación amigable para el usuario en la cual pueda ingresar sus distintos puntos a viajar y así la aplicación le dé la ruta más óptima para viajar hacia ella.
- Diseñar una interfaz que muestre la información de esas rutas y permita a su vez dar información más detallada sobre el cobro y tiempo de la carrera si desea hacer ese viaje en taxi.
- Poder simular lo que sería la parte de la llamada o uso de una unidad para llegar a un destino del usuario.
- Poder mostrar el costo aproximado en base a la distancia del costo total de una carrera dependiendo la tarifa que rija a esa hora en taxi.

4. Objetivos Académicos

- Poder asociar los conocimientos adquiridos en programación 2, Análisis y Diseño de Software y a lo larga de las distintas clases que se han tenido hasta este momento para poder hacer uso de dicho conocimiento más en si sobre las estructuras de datos en Java y poderlas aplicar en el proyecto.
- Poder usar diferentes librerías que nos ayuden a la construcción del proyecto las cuales nos brindan herramientas para lograr esto, para lograr esto en el proyecto se usara el Api de Google Maps para el desarrollo de la aplicación.
- Implementar una estructura por capas en el proyecto y así poder lograr usar todos los beneficios que conlleva hacer esto.

5. Programación del Proyecto

5.1. Estructura de Datos

En la programación se usó una estructura proporcionada por el api colección de Java la cual es ArrayList la cual es usada múltiples veces para poder guardar distintos datos encadenados que son arrojados por la aplicación.

Árbol de expansión mínima esta estructura se detalla más en la sección 5.7.

5.2. Programación por capas.

La programación por capas al ser una técnica orientada a la Programación Orientada a objetos fue implementada en el desarrollo del proyecto MoviFast.

5.2.1. Capa Presentación

Esta capa es donde se encuentra las interfaces con las cuales el usuario interactuara, los cuales fueron desarrolladas lo más minimalistas posible para evitar con función por parte del usuario.

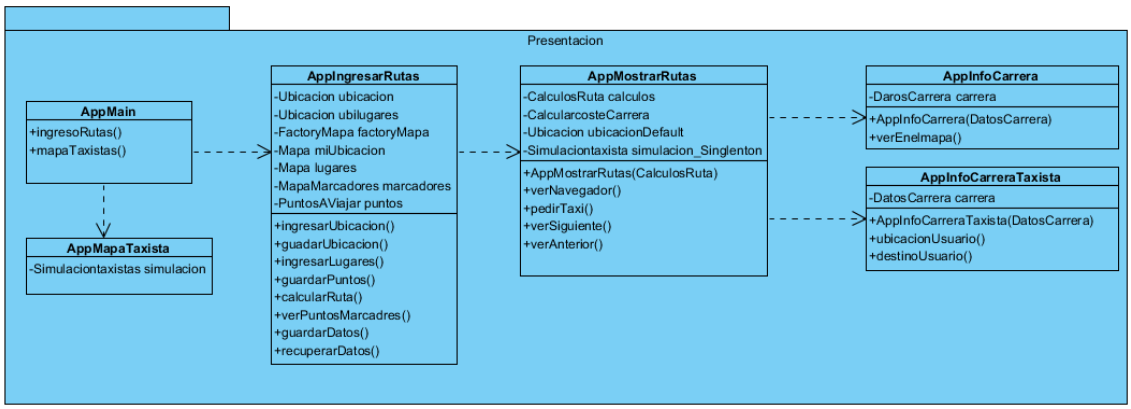

Fig. 2: Diagrama de Clases de la Capa de Presentación

AppMain: Interfaz gráfica la cual da la bienvenida al usuario y la cual le permite ingresar a la interfaz en la cual se ingresará las distintas rutas, así como poder acceder a un mapa de taxista el cual esta vez ha sido simulado por el sistema.

AppMapaTaxistas: Interfaz la cual nos permite ver la ubicación de taxistas simulados en el programa.

AppIngresarRutas: Interfaz la cual permite ingresar al usuario los distintos puntos a viajar y además le da varias opciones para obtener una mayor precisión al obtener estos datos.

AppMostrarRutas: Interfaz la cual se encarga de mostrar las diferentes rutas que se obtuvieron atreves del cálculo y obtención de un árbol de expansión mínima.

AppInfoCarrera y AppInfoCarreraTaxista: Interfaces la cuales se encargar de mostrar la información del viaje.

5.2.2. Capa de Lógica

Cerebro de la aplicación la cual se encarga de procesar todas las peticiones de la capa de presentación.

5.2.2.1. Paquete Cálculos

Esta se encarga de realizar los cálculos para obtener la ruta mínima, así como el cálculo del coste de la carrera a los distintos puntos de viaje.

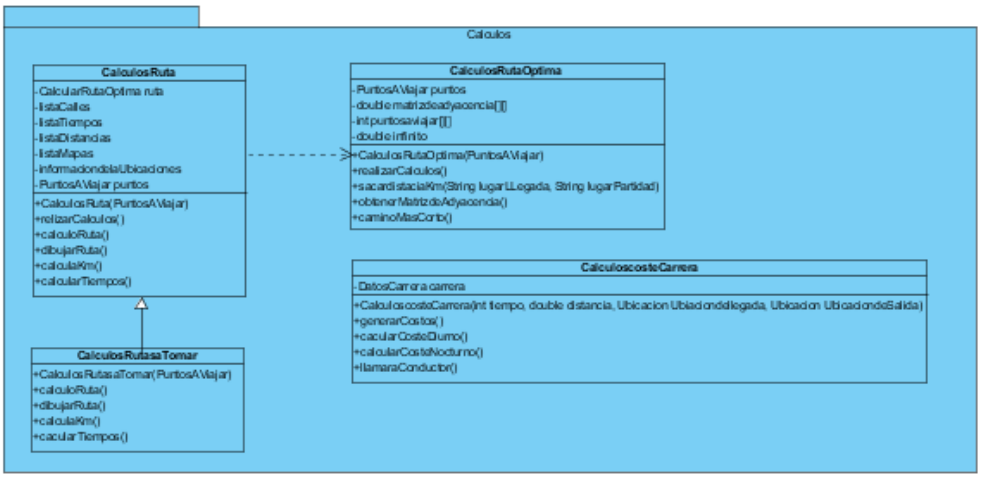

Fig. 3: Diagrama de Clases del paquete Cálculos

CalculosRuta: usa el Patrón de diseño template method el cual define un método realizarCalculos() método final el cual define un esqueleto que siempre debe cumplirse para obtener el cálculo de las rutas en la aplicación.

> calcularRuta(): método el cual se encarga de llamar al método de caminoMasCorto de la clase cálculosRutaOptima en el cual se obtiene un árbol de expansión mínima.
>
> dibujarRuta(): método el cual obtiene las distintas imágenes de las rutas que se mostraran en la capa de presentación.
>
> calcularKm(): método el cual me obtiene la distancia mínima para viajar por cada ruta que se obtuvo en el anterior calcularRuta.
>
> calcularTiempos(): método el cual me obtiene los tiempos mínimos para viajar por cada ruta que se obtuvo en calcularRuta.

Estos últimos 4 métodos son métodos abstractos.

CalculosdeRutaaTomar: clase la cual extiende de CalculosRuta e implementa los direntes métodos abstractos de la clase padre.

CalculosRutaOptima: clase la cual se encarga de obtener los distintos caminos del árbol de expansión mínima, para poder viajar a todos los puntos con un coste mínimo una sola vez (se profundizará más este tema adelante).

CalculosCosteCarrera: clase la cual se encarga de obtener:

- Valor de un viaje a dos puntos con tarifa diurna.
- Valor de un viaje a dos puntos con tarifa nocturna.
- Mostrar el tiempo de espera, llegada y distancia de un viaje, los dos últimos cálculos obtenidos al momento de realizar los cálculos de las rutas.

5.2.2.2. Paquete mapas

Este paquete contiene las clases las que se encargan de la interacción con el api de Google Maps para así obtener distintos tipos de mapas y a su vez poder interactuar con ellos.

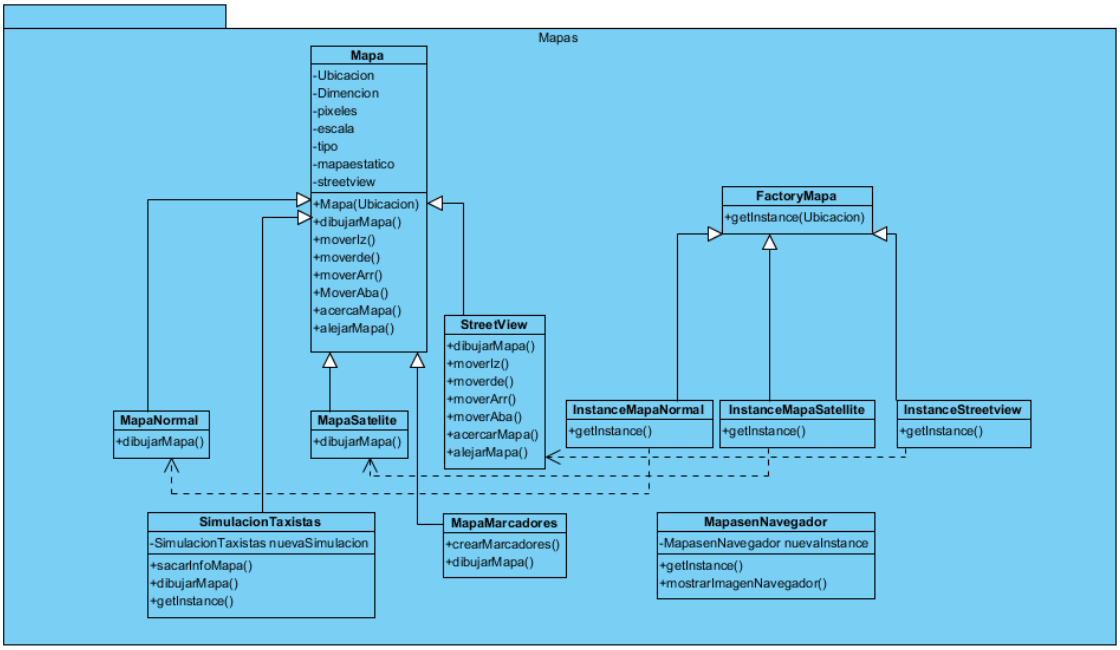

Fig. 4: Diagrama de clases de la Capa de Mapas

Mapa: clase abstracta la cual se encarga de definir métodos en común entre sus clases hijas los cuales nos sirven para poder manipular los distintos tipos mapas que serán mostrados en la aplicación.

dibujarMapa(): método abstracto el cual permitirá a las clases hijas definir qué tipo y como mostrar el mapa que crearan dichas clases.

MapaNormar, MapaSatellite, StreetView: clases las cuales nos permiten obtener los 3 diferentes tipos de mapa que están disponibles en el api de Google Maps están clases implementan el método dibujarMapa a su manera.

SimulacionTaxistas: Clase la cual se encarga de crear una simulación de taxistas en un mapa de Cuenca y así poder visualizar como en un futuro planea ser la aplicación ya implementada en la ciudad (Clase en la cual se implementó el Patrón de diseño singlenton para garantizar una instancia única y así obtener una única simulación en el sistema).

MapaMarcadores: Clase la cual nos crea un mapa en el cual se nos muestra marcadores de los puntos que el usuario planea visitar.

MapasenNavegador: Clase la cual nos devolverá un URL con una ruta que el usuario pide que se le muestre en el navegador predeterminado en el pc en el que se esté usando (Esta clase implementa el patrón de diseño singlenton ya que solo se necesita una instancia de dicha clase).

FactoryMapa: Interfaz la cual define el método getInstance(Mapa) el cual nos retornara una instancia de las 3 que heredan de la clase mapa.

5.2.3. Capa de Datos

Capa la cual se encarga de guardar, recuperar y eliminar datos los cuales van a ser usados en las otras capas.

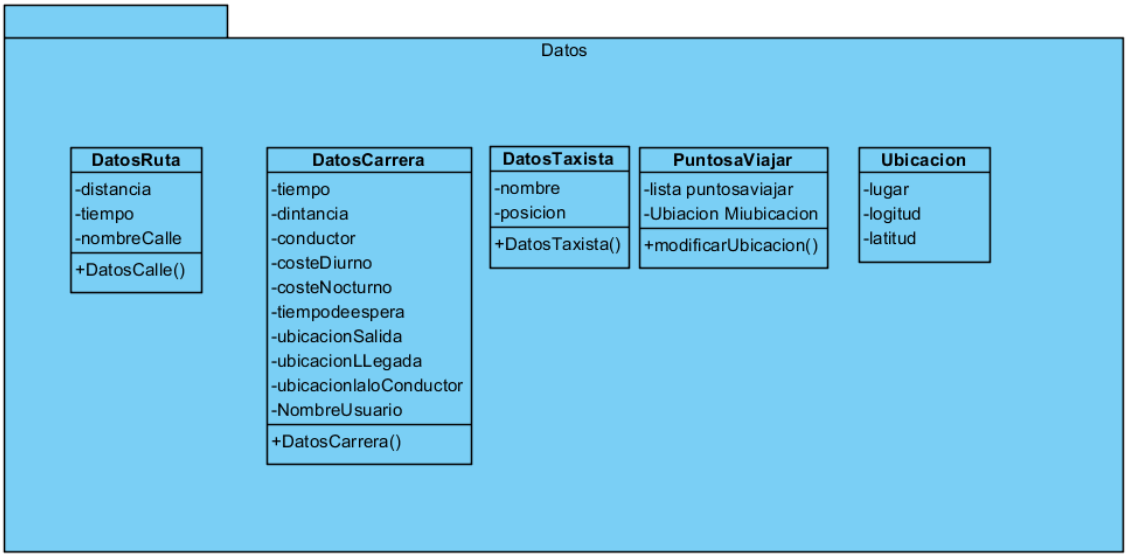

Fig. 5: Diagrama de Clases de la Capa de Datos

DatosRuta: Se encarga de guardar los diferentes datos que se obtienen de una ruta como es la distancia, tiempo, y el nombre de calle en la que se esté viajando.

DatosCarrera: Se encarga de guardar la información de los datos de un viaje en taxi como son el costo del viaje, el nombre de conductor, nombre del pasajero, la ubicación de salida y llegada, el tiempo y la distancia del viaje.

DatosTaxi: Se encarga de guardar los datos de un taxista en este caso solo su nombre y su posición.

Ubicación: Se encarga de guardar la información de un lugar a viajar como es su nombre y sus coordenadas.

PuntosaViajar: Se encarga de guardar los distintos puntos a los que se desea viajar.

5.3. Diagrama de Clases

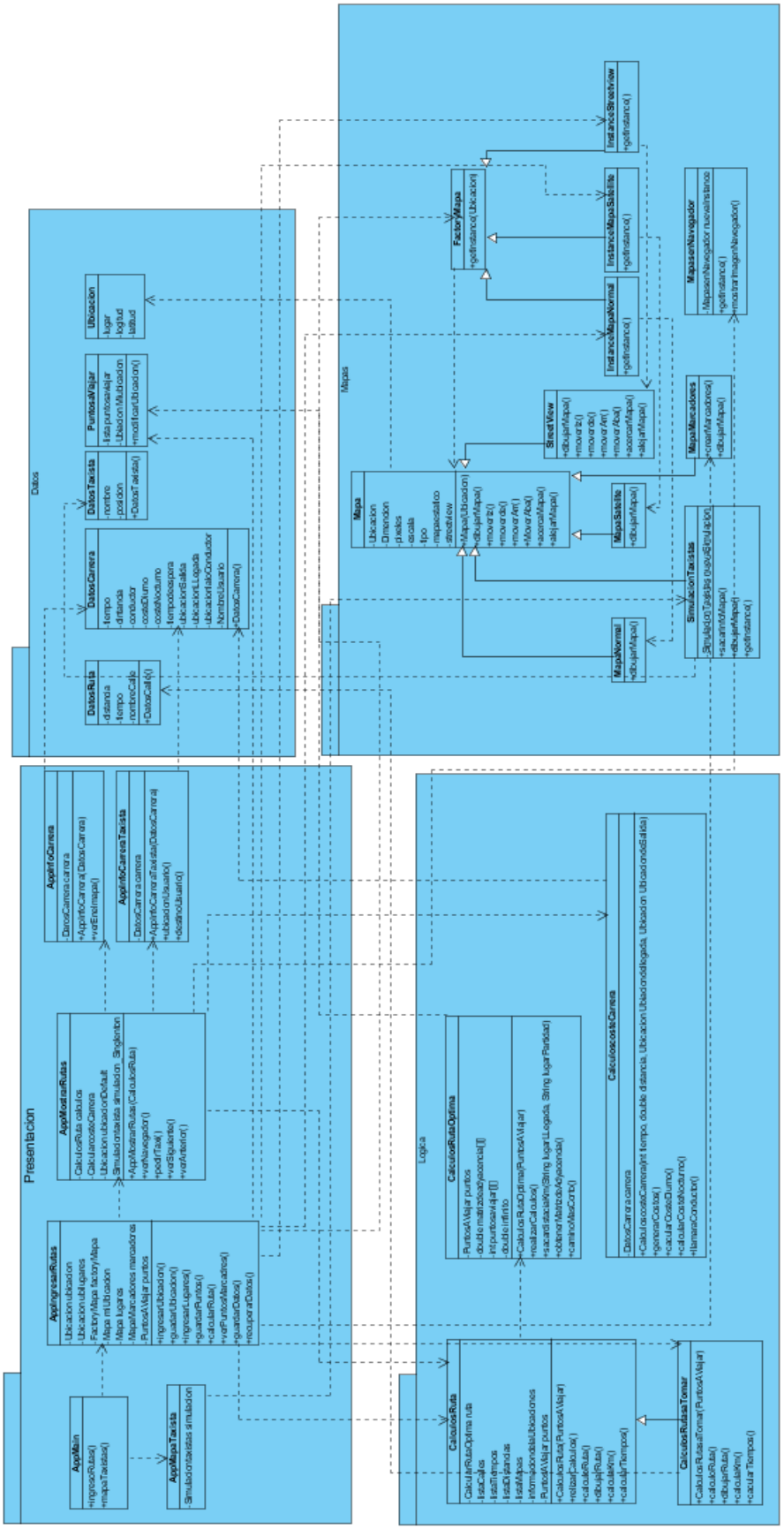

Fig. 6: Diagrama de clases del proyecto MoviFast

5.4. Interfaz gráfica de usuario

El sistema provee una interfaz gráfica que busca ser lo más intuitiva posible para facilitar el uso al usuario en esta sección se mostrara las distintas partes que componen la interfaz de MoviFast.

5.4.1. Ventana de Inicio

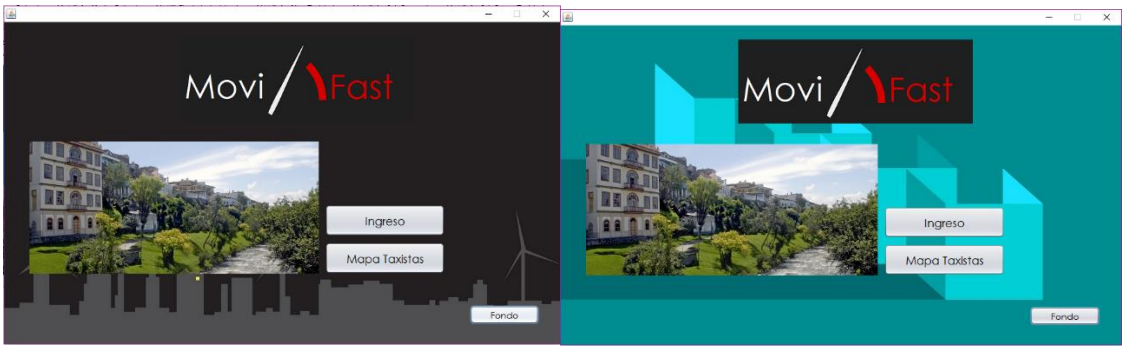

Fig. 7: Ventana de inicio de la aplicación.

Le da la bienvenida al usuario que al momento de elegir la opción de ingreso la cual no lleva a la ventana de ingreso de las rutas o la opción de poder ver el mapa de los taxistas que el sistema está simulando.

5.4.2. Ventana para el Ingreso de la Rutas

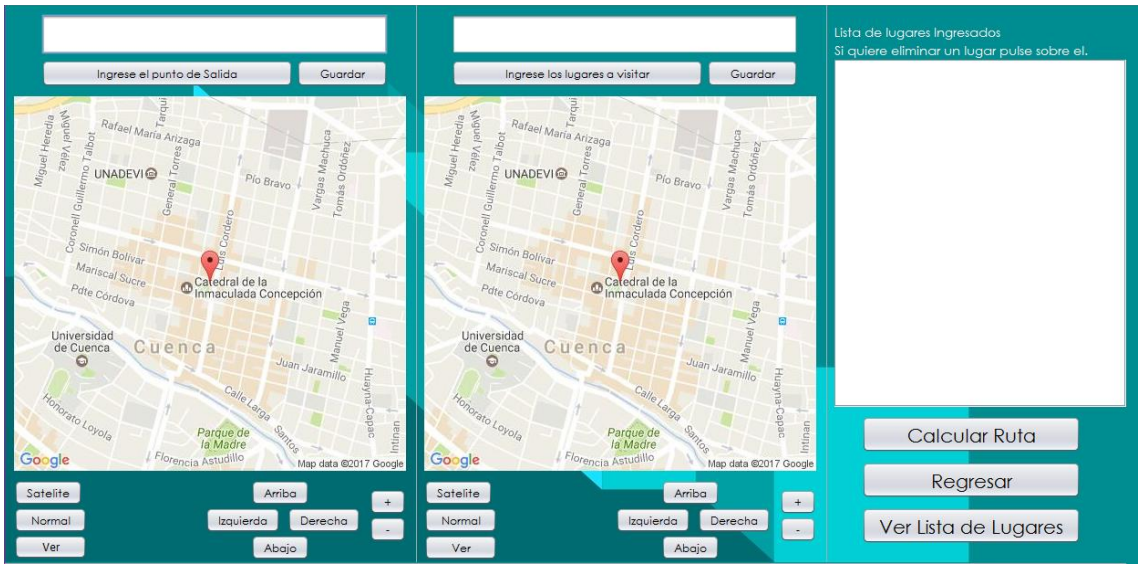

Fig. 8: Ventana para el ingreso de rutas en la aplicación

Esta interfaz es en la cual el usuario va interactuar para poder guardar los diferentes puntos a los que desea viajar, además esta interfaz cuenta con varias herramientas para la máxima precisión por parte del usuario.

Una vez ingresado las rutas el usuario puede o calcular las rutas óptimas para el viaje o puede ver en un mapa los distintos puntos ya ingresados.

5.4.3. Ventana para mostrar las rutas a viajar

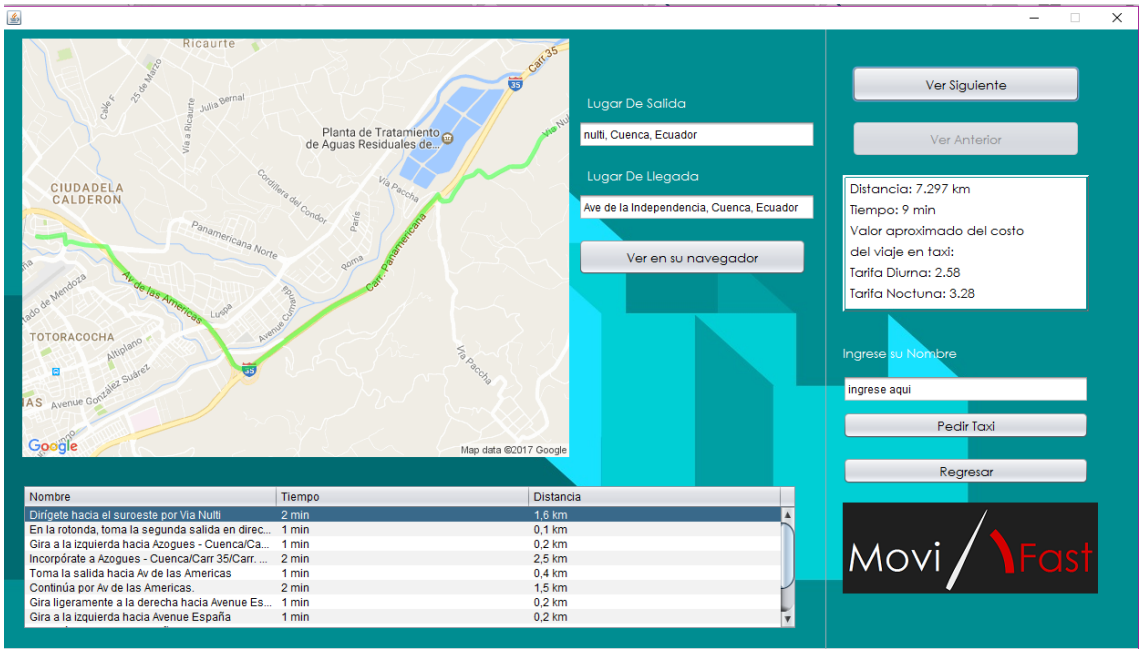

Fig. 9: Ventana para Mostrar las Diferentes Rutas.

Esta interfaz se encarga de mostrar las diferentes rutas obtenidas mediante el cálculo hecho por la aplicación a la ves de mostrar información sobre el costo del taxi e información de la ruta.

5.4.4. Ventanas para mostrar marcadores en un mapa

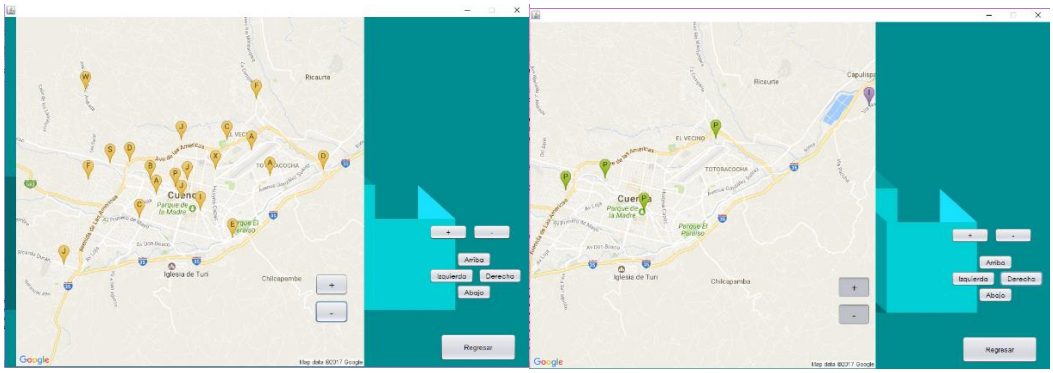

Fig. 10: Ventanas que me muestran Izquierda: Ubicación Taxista Derecha: Puntos a Viajar

Estas interfaces se encargan de mostrar al usuario las ubicaciones de la simulación de los taxistas y también de los distintos puntos a visitar que fueron ingresados.

5.4.5. Ventanas para mostrar información del viaje en taxi

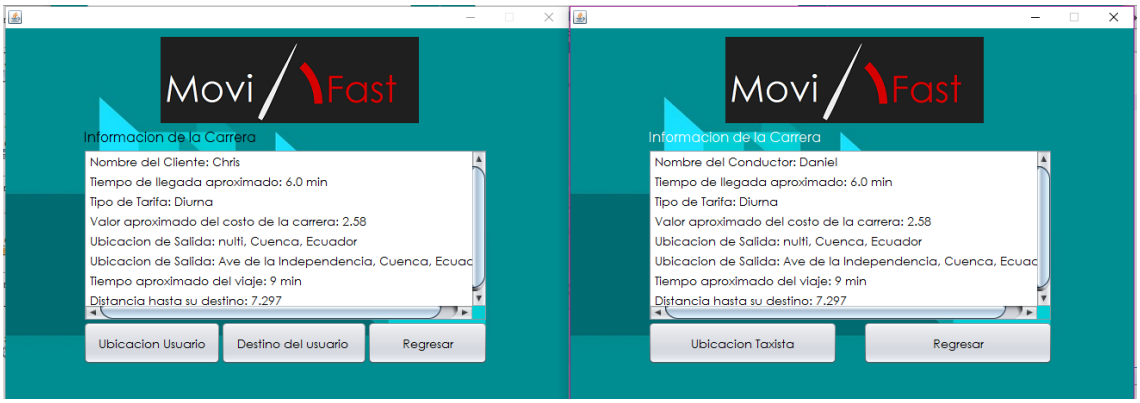

Fig. 11: Ventanas que muestran información del viaje en taxi. Izquierda: Interfaz Taxista Derecha: Interfaz Cliente

Las ventanas para mostrar la información de un nuevo viaje en Taxi en la cual al usuario le muestra el taxista más cercano, el tiempo aproximado para que llegue dicho taxista, información del viaje y el coste del viaje, cabe decir que el sistema cuenta con la capacidad para diferenciar entre la tarifa diurna de la nocturna.

5.5. Api de Google Maps

Google pone a disposición de los desarrolladores su api la cual está disponible para sistemas operativos Android, IOS, Web, gracias a una librería creada para el uso de api en Java se pudo adaptar está a la aplicación MoviFast la librería de nombre MapsJava nos permite usar varias herramientas del api de Google Maps las cuales vamos a hablar a continuación.

5.5.1. Static Maps

https://maps.googleapis.com/maps/api/staticmap

Google nos ofrece la capacidad de obtener una imagen de un mapa estático el cual va a ser personalizado por nosotros deberemos ingresar la ubicación de un punto o varios en el mapa, el tamaño de la imagen, así como el acercamiento del mapa, y que tipo de mapa queremos que se nos muestre una vez que se tienen todos estos parámetros se tiene algo como esto:

https://maps.googleapis.com/maps/api/staticmap?center=-2.899169,-78.98440100000002&zoom=13&size=640x640&maptype=roadmap

Con este URL se obtiene una imagen de un mapa estático devuelto por el api de Google Maps el cual puede obtenerse en java y mostrarse en una interfaz gráfica o se puede hacer uso de la librería MapsJava el cual en la clase StaticMap hay un método el cual tiene la finalidad de obtener dicha imagen del mapa.

```
public Image getStaticMap(String centerAddress, int zoom, Dimension size, int scale, Format format, Maptype maptype)
```

Fig. 12: Método para obtener una imagen estática de una parte de un Mapa.

Como ya se había indicado para obtener la imagen de un punto en el mapa se deben ingresar ciertos parámetros los cuales son:

centerAddress: En esta variable se guarda el punto en el que se desea centrar el mapa puede ingresarse el nombre del lugar, así como sus coordenadas

zoom: En esta variable de guarda con cuanto acercamiento queremos la imagen el api de Google como máximo y mínimo acercamiento limita los valores entre 0 y 21 niveles de zoom.

size y scale: La scale o escala nos permite decir cuál es el tamaño máximo de la imagen que queremos obtener y la variable size nos dice cuál es la dimensión de esa imagen.

scale = 1: 640 * 640 máximo

scale = 2: 640 * 640 máximo. pero devuelve más pixeles en la imagen.

Además, hay otras escalas, pero esas forma parte Premium del api de Google Mapas.

format: esta variable es para decir el formato de imagen que queremos que el api nos devuelva hay varias acciones para elegir gif, png, jpeg.

mapType: nos permite decir que tipo de mapa queremos que nos devuelva el api de Google Maps, en el proyecto se usó solo mapas de tipo normal o roadmap y por satélite o terrain.

Para más información se puede visitar la página:

https://developers.google.com/maps/documentation/static-maps/intro?hl=es-419

5.5.2. Route

En el proyecto se hace uso de clase route de la librería MapsJava el cual tiene un método llamado.

```
public String[][] getRoute(String originAddress, String destinationAddress, ArrayList<String> waypoints, Boolean optimize, mode travelMode,
```

Fig. 12: Método para generar una ruta ingresando dos puntos.

El método getRoute es utilizado para obtener la información de los distintos puntos que componen una ruta en este método se debe agregar como puntos necesarios el punto de salida y el punto de llegada de una ruta para así poder obtener dicha información además debemos agregar el tipo de viaje que realizaremos los cuales pueden ser.

Driving: Nos devuelve la ruta más óptima por carretera.

Walking: Nos devuelve la ruta más óptima viajando a pie.

Bycycling: Nos devuelve la ruta más óptima usando ciclovías.

Además, podremos mandar una lista de lugares que se deben pasar si o si por dicha ruta, así como evitar ciertas restricciones viales.

La librería nos ofrece la capacidad de obtener la distancia mínima de una ruta, así como el tiempo aproximado de viaje.

```
public ArrayList<Integer> getTotalTime() {
public ArrayList<Integer> getTotalDistance() {
```

Fig. 13: Métodos para obtener la distancia mínima y tiempo de viaje.

5.5.3. Geocoding

Así también el api de Google Maps nos da la capacidad de obtener las coordenadas de un lugar de acuerdo al nombre de dicho lugar y viceversa.

```
private final String URLRoot = "http://maps.google.com/maps/api/geocode/xml";
private final String pathStatus = "GeocodeResponse/status";
private final String pathPostalcode = "GeocodeResponse/result/address_component";
```

Fig. 14: Variables las cuales nos enlazan con el api de Google para obtener la información de un punto en el mapa.

Nuevamente usando la librería MapsJava esta vez la clase Geocoding se podrá obtener información de un punto del mapa que mandamos como parámetro en los siguientes métodos.

```
getCoordinates(String address)

getAddress(Double latitude, Double longitude)
```

Fig. 15: Métodos para obtener las coordenadas de un lugar y a su vez con las coordenadas obtener el nombre de un lugar.

getCoordinates(String address): Método el cual nos devuelve la coordenadas en longitud y latitud de una dirección que es mandada como parámetro.

getAddress(): Método el cual recibe la coordenadas de un lugar en latitud y longitud y nos devuelve el nombre de ese lugar.

5.5.4. StreetView

El api de Google nos permite el acceso a una imagen de un sitio en este caso usando el URI directamente o al igual que los casos anteriores la librería MapsJava nos proporciona la clase StreetView la cual tiene un método para obtener una imagen de ese lugar.

```
public Image getStreetView(String address, Dimension sizeImage, double heading, double fov, double pitch)
```

Fig. 16: Método para obtener una imagen de SteetView de un lugar.

Heading: nos permite definir el ángulo del cual queremos obtener la imagen (0 - 360).

Fov: se encarga de controlar la profundidad del lugar (0 - 120)

Pitch: se encarga de controlar el ángulo vertical de la imagen a obtener (0 -120)

5.5.5. Google Maps Web

En el proyecto se usa la página web de Google Maps para poder mostrar información de las rutas las cuales son mostradas en el navegador principal de la Pc en donde se corra el sistema.

https://www.google.es/maps/dir/-2.8976354,-79.0279857/-2.9044418,-79.0030819/@-2.9073668,-79.0323931,13z/data=!4m2!4m1!3e0

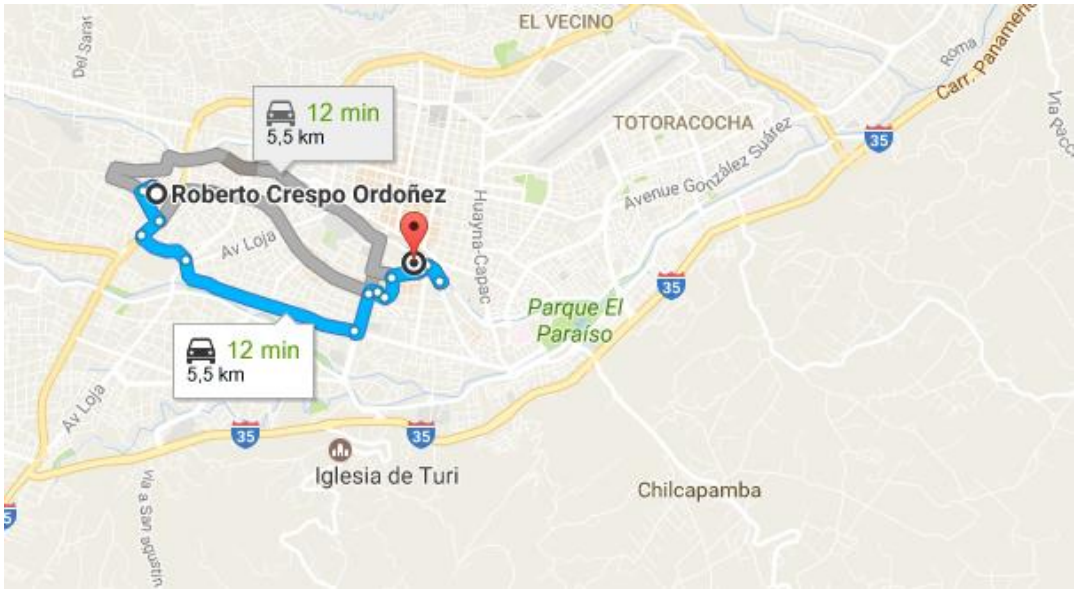

Fig. 17: Imagen de la página de Google Maps mostrando una ruta solicita en la aplicación MoviFast

```java
public void mostrarImagenNavegador(String ubciacion1 , String Ubicacion2)
{
    String urlweb = "https://www.google.es/maps/dir/";
    urlweb += ubciacion1 +"/"+Ubicacion2 +"/@-2.9073668,-79.0323931,13.13z/data=!4m2!4m1!3e0";
    try {
        URI urlmaps = new URI(urlweb);
        Desktop.getDesktop().browse(urlmaps);
    } catch (Exception e) {
        System.out.println("ocurrio un problema al abrir el navegador");
    }
}
```

Fig. 18: Código para poder abrir una página web en java

En la figura 18 se puede observar el método el cual nos permite abrir el navegador y a su vez mandar a que se habrá la página de Google Maps la cual nos mostrará la ruta con los 2 puntos que mandamos como parámetro.

5.6. Librería Jsoup.

Es una librería que nos permite pasar de código HTML a código el cual se pueda entender por parte del usuario esto se utiliza al momento que creamos una ruta con el método getroute ver figura 12. El cual nos devuelve la información de la ruta o de las calles por la que pasara esa ruta la cual nos devuelve en formato HTML el cual gracias método parse de la librería se transformara dicho código en código leíble para el usuario.

```java
//Transoforma HTML a texto
nombreCalle = Jsoup.parse(caminos[j][2]).text();
```

Fig. 19: Código en el cual se transforma el nombre de una calle obtenida por el api del Google a información leíble.

5.7. Árbol de expiación mínima

El proyecto usa un árbol de expiación mínima para poder obtener el camino más corto viajando por todos los nodos, para lo cual al principio se planeó usar el algoritmo de Prim para poder encontrar dichos caminos, pero de este se obtuvo un problema el cual el árbol en ocasiones da caminos los cuales tienen que pasar 2 o más veces por un mismo nodo veces por un mismo nodo.

Esto se puede apreciar en la fig. 20. En la cual muestra un grafo el cual al hacer el algoritmo de Prim se obtiene el árbol de expiación mínima de ese grafo, pero el nodo se queda conectado a los nodos B y C lo cual, en la vida real, para el ejemplo de obtener rutas mínimas por carretera significaría o bien desde C viajar a B o D primero, regresar a C y viajar al nodo que aún no se ha visitado lo cual no está correcto porque se necesita visitar solo una vez el nodo C.

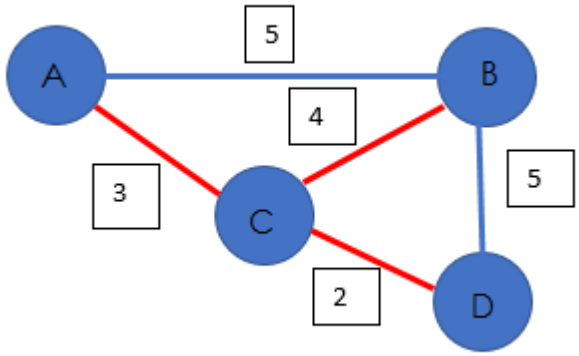

Fig. 20. Grafo al cual por el algoritmo de Prim se obtuvo el árbol de expiación mínima.

Como podemos ver el grafo obtenido por el algoritmo de Prim no nos da una solución óptima para el uso en la aplicación MoviFast ya que como se ve en la fig. 21 el árbol real debería ser un árbol el cual cada nodo solo se pueda unir como máximo a dos nodos y pasar por todos los nodos una solo vez, por lo cual se realizó una modificación al algoritmo de Prim para lo cual se siguieron los siguientes pasos.

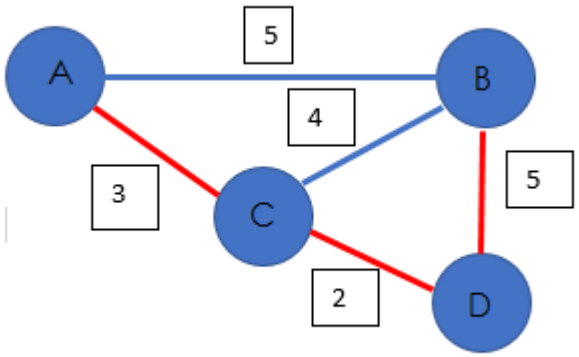

Fig. 21: Árbol de expiación mínima pasando por todos los nodos una vez.

1. El nodo en el cual se va a partir y el ultimo solo se puede unir a un nodo el cual tenga la menor distancia para visitar estos.
2. Los demás como máximo se podrán unir a dos nodos por los caminos mínimo de cada nodo.

Una vez hecho esto se obtiene el grafo de la fig. 21.

5.7.1. Implementación en Java.

```
public int[][] caminoMasCorto() {
    obtenerMatrizAdyacencia();
    int vecto[] = new int[getMatrizdeadyacencia().length];
    vecto = llenarvecto(vecto);//este vector me llevara la cuenta de a cuantos nodos se conecta el nodo que se esta visitando
    int contadorVertices = 0;
    int[][] matrizCaminos = new int[getMatrizdeadyacencia().length-1][2];//Aqui se guardara los distintos camininos que se obttengan
    int filaCaminos = 0;
    final int vertices = getMatrizdeadyacencia().length;
    int puntoSalida = 0;//punto salida
    int puntoLlegada = 0;//punto llegada
    double minimo;
    while (contadorVertices < vertices) {
        minimo = getInfinito();
        for (int i = 0; i < vertices; i++) {
            if(puntoSalida == i)
            {
                matrizdeadyacencia[puntoSalida][i] = infinito;
            }
            else
            {
                if(matrizdeadyacencia[puntoSalida][i] < minimo && vecto[puntoSalida] < 2 && vecto[i] < 2)//aqui se obtendra el valor de el camino mas c
                {                                                                          //y cuando el nodo visitado  no se conecten a 2 nodos
                    minimo = matrizdeadyacencia[puntoSalida][i];
                    puntoLlegada = i;
                }
            }
        }
        vecto[puntoSalida]++;
        vecto[puntoLlegada]++;

        contadorVertices++;
        System.out.println("El Vertrice : " + puntoSalida + " , " + puntoLlegada + " Costo de " + minimo);
        matrizCaminos[filaCaminos][0] = puntoSalida;
        matrizCaminos[filaCaminos][1] = puntoLlegada;
        filaCaminos++;
        puntoSalida = puntoLlegada;

        getMatrizdeadyacencia()[puntoSalida][puntoLlegada] = getInfinito() + 100;
        getMatrizdeadyacencia()[puntoLlegada][puntoSalida] = getInfinito() + 100;

        if (filaCaminos == getMatrizdeadyacencia().length - 1) {

            return matrizCaminos;
        }

    }

    return matrizCaminos;
}
```

Fig. 22: Árbol de expiación mínima pasando por todos los nodos una vez.

En la figura 22 podemos ver la implementación en java de el algoritmo modificado de Prim el cual obtiene un árbol de expiación minina pasando por todos los puntos.

5.8. Patrones de diseño usados en el proyecto

5.8.1. Factory Method

El patrón de diseño Factory se usa para poder obtener instancias de los 3 diferentes tipos de mapa que son utilizados del api de Google Maps ya que al ser una familia de objetos que tiene en como la clase padre Mapas la forma más correcta para acceder a instancias de estas clases son mediante este patrón.

En la fig. 23 se puede observar mediante un diagrama de clases el uso de este patrón en el proyecto.

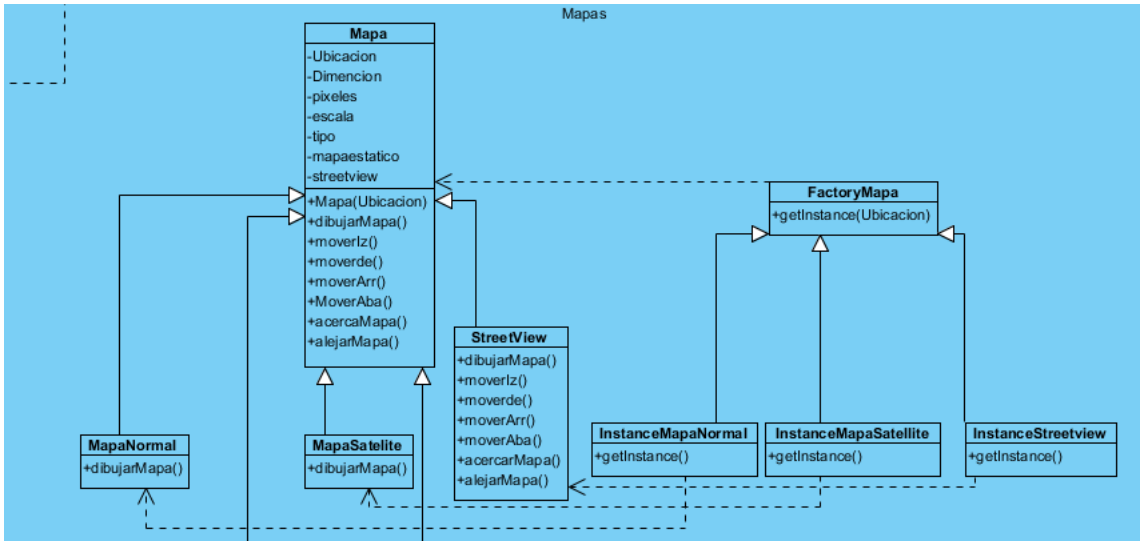

Fig. 23: Patrón de diseño Factory implementado en el Proyecto MoviFast

5.8.2. Template Method

El patrón de diseño template fue utilizado en el proyecto para definir los pasos a seguir para calcular ruta y la información de ellas.

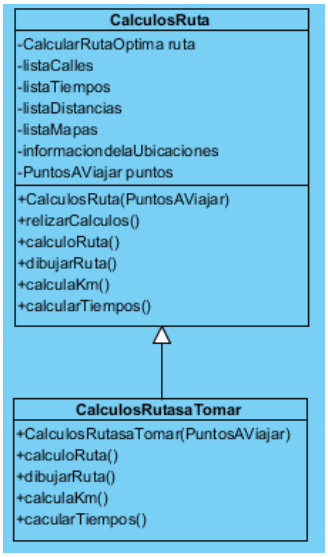

Fig. 24: Patrón de diseño Template implementado en el Proyecto MoviFast

En el proyecto el método realizarCálculos() de la clase CalculosRuta es un método final el cual se encargara de realizar los pasos a seguir para obtener los cálculos de la ruta.

los métodos calculoRuta(), dibujarRuta(), calcularKm(), calcularTiempos() los cuales son definidos en la clase CalculosRuta serán implementados por la clase CalculosRutaaTomar.

```
public CalculosRuta(PuntosAViajar puntos) {
    this.puntos = puntos;
}

public  final void realizarCalculos() {
    calculoRuta();
    dibujarRuta();
    calcularTiempos();
    calculaKm();

}
public abstract void calculoRuta();
public abstract void dibujarRuta();
public abstract void calcularTiempos();
public abstract void calculaKm();
```

Fig. 25: Patrón de diseño Template implementado en Java del Proyecto MoviFast

5.8.3. Singlenton

Este patrón de diseño se encarga de crear instancias únicas de las clases SimulacionTaxistas y MapasenNavegador ya que estas dos clases dentro del proyecto deberían ser instanciadas una solo ves para garantizar esta única instancia se utiliza este patrón.

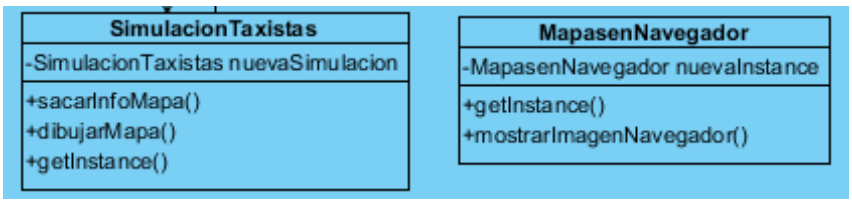

Fig. 26: Patrón de diseño Singlenton implementado en el Proyecto MoviFast

5.9. Tarifas de taxi en Cuenca en el proyecto MoviFast

Desde el año 2014 en Cuenca el uso del taxímetro es obligatorio en la ciudad por lo cual se pusieron ciertos parámetros para obtener el valor final de un viaje en taxi, esos valores parámetros fueron puestos por un estudio hecho por la Universidad de Cuenca y modificados por última vez en el año 2015.

El valor de un viaje en taxi según estos parámetros queda:

Tarifa Diurna 5:00 – 22:00:

Valor de inicio de la carrera:

Arranque: 55 centavos.

Valor del Kilometro: 29 centavos.

Valor pasado los 7 kilómetros: 36 centavos.

Valor por minuto de espera: 6 centavos.

Tarifa Nocturna 22:00 - 5:00:

Valor de inicio de la carrera:

Arranque: 55 centavos.

Valor del Kilometro: 39 centavos.

Valor pasado los 7 kilómetros: 46 centavos.

Valor por minuto de espera: 6 centavos.

De acuerdo a estos valores la aplicación hace un cálculo para obtener el valor aproximado de un viaje en taxi en la figura 26 y 27, se puede observar la implementación de la tarifa diurna y nocturna en java.

```
private void calcularCosteDiurno()
{
    double coste = 0;//incializamos el valor de la ruta
    double km = Math.floor(getCarrera().getDistaciaaViajar());//obtenemos la parte entera de la distancia de la carrera
    double sobrante = getCarrera().getDistaciaaViajar() - km;//asi tambien se obtine la parte que sobra en m para calcular su costo
    double costoSobrante;
    if(km < 7)//si el la distancia de la ruta es menor a 7 km
    {
        coste = km * 29 + 55;
        costoSobrante = (sobrante*29);
        coste = coste + costoSobrante;
        coste = redondeo(coste/100);

    }
    else//si la distancia es mayor a 7
    {
        double kmaux = km - 7;
        coste = 7 * 29 + 55 + kmaux*36;
        costoSobrante = (sobrante*36);
        coste = coste + costoSobrante;
        coste = redondeo(coste/100);

    }

    if(coste < 1.39)//si los valores obtenidos son menores a la carrera minima esta toma ser el coste final.
    {
        coste = 1.39;
    }
    getCarrera().setCostoDiurno(coste);
}
```

fig. 26: Código que implementa el caculo del coste de una carrera diurna en Cuenca

```
private void calcularCosteNocturno()
{
    double coste = 0;
    double km = Math.floor(getCarrera().getDistaciaaViajar());
    double sobrante = getCarrera().getDistaciaaViajar() - km;
    double costoSobrante;
    if(km < 7)
    {
        coste = km * 39 + 55;
        costoSobrante = (sobrante*39);
        coste = coste + costoSobrante;
        coste = redondeo(coste/100);

    }
    else
    {
        double kmaux = km - 7;
        coste = 7 * 39 + 55 + kmaux*46;
        costoSobrante = (sobrante*46);
        coste = coste + costoSobrante;
        coste = redondeo(coste/100);

    }
    if(coste < 1.67)
    {
        coste = 1.67;
    }
    System.out.println(coste);
    getCarrera().setCostoNocturno(coste);
}
```

fig. 27: Código que implementa el caculo del coste de una carrera nocturna en Cuenca

6. Conclusiones Y Recomendaciones

- Unos de los factores que intervienen negativamente en el proyecto es el tiempo de respuesta del api de Google Maps al momento de pedir información sobre las rutas y otras peticiones hacen que la aplicación en ocasiones se vuelva muy lenta.
- A veces la conexión con el api de Google se pierde por lo cual la aplicación realiza la petición recursivamente hasta que la petición se realice con éxito lo cual aumenta el tiempo de espera de peticiones.
- El api de Google, aunque en su página oficial ofrece un api actualizada con informes del tránsito en carretera, pero al momento de pedir estos datos siempre nos devuelve el valor por defecto o valor medio para llegar a esa ruta, en un futuro si se arregla esto o con una actualización se podrá obtener una estimación más precisa sobre los datos del camino.
- Como se vio el Proyecto en la parte para calcular los datos de la ruta se optó por realizar un template method el cual en un futuro podrá ayudar a implementar más opciones para el viaje como son las rutas a pie, ciclovía, o bus.

- Se tiene que decir que los valores de los costes realizados por la aplicación son valores aproximados, por lo cual el valor real podría variar un poco dependiendo de condiciones de viaje lo que sí se puede decir con certeza que el valor obtenido por la aplicación ese valor se aproxima muchísimo al real tomando en cuenta la distancia a viajar.

7. Referencias

Joyanes, Aguilar. (2008), Estructura de datos en Java. Madrid, España: Editorial Mac Gran Hill

Redacción De Diario El Tiempo (2015) "Tarifas de taxi suben de costo en carreras largas" Diario El Tiempo. Cuenca octubre 2015.

Valladarez, David. (2016), Taxi Compartido. Disponible en: https://es.scribd.com/document/331356956/ProyectoMAPS. Cuenca Ecuador.