Algorithm design

In the following paragraph are presented the most relevant and significant algorithm used in Travlendar+ application.

More specifically are described the following algorithm:

* Creation of new appointment;
* View of daily schedule;
* Check appointments overlapping;
* Check appointments unreachability;
* Check travel alternatives;

The application is written in Java.

Before to explain the algorithms details, it is needed to introduce the class Appointment and Travel that are the most important entities for the whole application.

## Appointment class

The class Appointment has as attributes:

* The appointment name;
* The date;
* The time when it starts;
* Address of destination;
* Address of departure;
* The desired time to leave;
* The expected time of arrival;
* The appointment duration;
* The associated travel to reach the appointment;

public class Appointment **{**

String name**;**

Date date**;**

Time beginTime**;**

Address destination**;**

Address departure**;**

Time departureTime**;**

float arrivalTime**;**

float duration**;**

Travel travel**;**

**}**

## Travel Class

The travel class has the following attributes:

* The related appointment;
* The departure point;
* The destination point;
* The travel type;
* The movements list in which the travel is split;

public class Travel **{**

Appointment appointment**;**

Address departure**;**

Address destination**;**

TravelType travelType**;**

Weather weatherCondition**;**

ArrayList**<**Movement**>** movements**;**

public float computeTravelTime**();**

public float computeTravelCost**();**

**}**

There are two methods in the travel class:

* computeTravelTime() allows to compute the travel duration.
* computeTravelCost() allows to compute the travel cost. Cost includes car sharing, gasoline, toll, tickets and so on.

These methods are used in faster/cheaper selection.

# Other data structures

enum TravelMode **{** DRIVING**,** WALKING**,** BICYCLING**,** TRANSIT**;** **}**

enum TravelType **{** FASTER**,** CHEAPER**,** GREEN**,** ONLYOWNCAR**,** ONLYPUBLICTRANSPORT**;** **}**

enum MovementMean **{** CAR**,** WALK**,** PUBLICTRANSPORT**,** BIKE**,** CARSHARING**,** BIKESHARING**;** **}**

the enumeration TravelMode represents the all possible travel modality that the Google Maps API supports.

TravelType is the enumeration that contains all kind of travel provided by the application.

MovementMean, instead of other two, contains all alternatives for each movement.

# Create new appointment

This algorithm allows the user to create a new appointment.

In this algorithm is used one method called readText, this is a method created to read from EditText and has as parameters only the id of EditText.

public Appointment createAppointment**()** **{**

Appointment appointment**;**

String name **=** readText**(**"name"**);**

Date date **=** formatDate**.**parse**(**readText**(**"date"**));**

Time beginTime **=** Time**.**valueOf**(**readText**(**"beginTime"**));**

Address destination **=** **new** Address**(**readText**(**"destination"**));**

Address departure **=** **new** Address**(**readText**(**"departure"**));**

Time departureTime **=** Time**.**valueOf**(**readText**(**"departureTime"**));**

float arrivalTime **=** Float**.**parseFloat**(**readText**(**"arrivalTime"**));**

float duration **=** Float**.**parseFloat**(**readText**(**"duration"**));**

appointment **=** **new** Appointment**(**name**,**date**,**beginTime**,**destination**,**departure**,**departureTime**,**arrivalTime**,**duration**);**

**return** appointment**;**

**}**

# View of daily schedule

When the user clicks on view daily schedule button, the system computes the daily schedule through this method and show to the user all appointments expected for the selected day and all travel to reach them.

This method has two parameters, the first is the instance of the calendar and the second in the desired day.

The first step performed is one query to user calendar thus to extract all appointment for the inserted day.

For each appointment it is necessary to re-compute the related travel, therefore is possible to have updated information.

But it is necessary to control, for every appointment, the unreachability and the overlapping with the next one.

The algorithm starts from the first appointment and control its reachability, if it is then fetch the next one appointment and control its reachability, if it is also reachable then it is possible to control the overlapping between them.

If the second appointment is not reachable (and it is deleted) then the algorithm fetches the next one and control its reachability. This cycle goes on until “the next appointment” is reachable and it is possible to control the overlapping.

If overlap control is positive (return true) means that the first appointment overlaps with the second and user must choose which is the appointment to keep.

At the end of all controls, the system computes the travel for the first appointment.

Then repeat all these steps for each appointment expected for the selected day.

public void viewDailySchedule**(**Calendar calendar**,** DateTime day**)** **{**

ArrayList**<**Appointment**>** appointmentList **=** **new** ArrayList**<**Appointment**>();**

Appointment appPrevious**;**

Appointment appNext**;**

appointmentList **=** calendarQuery**(**calendar**,** day**);**

**for(**int i**=**0**;** i**<** appointmentList**.**size**()-**1**;** i**++)** **{**

appPrevious **=** appointmentList**.**get**(**i**);**

appNext **=** appointmentList**.**get**(**i**+**1**);**

//needed for first cycle

**if(**checkUnreachability**(**appPrevious**))** **{**// if unreachble

appPrevious **=** manageUnreachability**(**appPrevious**);** //change parameters (and is reachable) or delete appointment

**if(!**appPrevious**.**equals**(null))**

appPrevious**.**travel **=** **new** Travel **(**appPrevious**.**departure**,** appPrevious**.**destination**,** preferences**.**travelMode**);**

**}**

**else** **{** // if reachable

**if(**checkUnreachability**(**appNext**))**

appNext **=** manageUnreachability**(**appNext**);** //change parameters (and is reachable) or delete appointment

**if(**appNext**.**equals**(null))** **{** //appNext is unreachable and is deleted

**do** **{**

i**++;**

appNext **=** **(**Appointment**)** calendarData**.**getEntries**().**get**(**i**);**

**if(**checkUnreachability**(**appNext**))**

appNext **=** manageUnreachability**(**appNext**);**

**}while(**appNext**.**equals**(null))**

**}**

//surely appNext is not null

**if(**checkOverlap**(**appPrevious**,** appNext**))**

appPrevious **=** manageOverlap**(**appPrevious**,**appNext**);** //choose which is the appointment to keep

**else**

appPrevious **=** appNext**;** //Save previous appointment and go on

appPrevious**.**travel **=** **new** Travel **(**appPrevious**.**departure**,** appPrevious**.**destination**,** preferences**.**travelMode**);**

**}**

**}**

**}**

# Check overlap

When the system has to check the overlapping between two appointments invokes this method that has as parameter the two appointments.

The method controls which appointment starts before and then controls if the begin time of the second appointment overlaps with end time of the first appointment.

The method returns false if there is not overlap and true otherwise.

public boolean checkOverlap**(**Appointment app1**,** Appointment app2**)** **{**

**if(**app1**.**beginTime**.**before**(**app2**.**beginTime**))** //appointment 1 is before 2

**if(**app2**.**beginTime**.**after**(**app1**.**beginTime**.**add**(**app1**.**duration**))**

**return** **false;** // no overlap

**else**

**if(**app1**.**beginTime**.**after**(**app2**.**beginTime**(**app2**.**duration**))**

**return** **false;** // no overlap

**return** **true;**

**}**

# Check unreachability

This method checks the unreachability of an appointment passed as parameter.

To check the unreachability it is necessary to control if the arrival time of one appointment is compatible with departure time added to estimated travel time.

The method returns false if the appointment is not reachable, true otherwise.

public boolean checkUnreachability**(**Appointment appointment**){**

**if(**appointment**.**arrivalTime**.**after**(**appointment**.**departureTime**.**add**(**appointment**.**travel**.**computeTravelTime**())))**

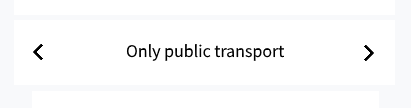
**return** **false;**

**return** **true;**

**}**

# Check travel alternative

The user can check all available travel alternatives and modify the actual one with another. This is possible switching the different alternatives.



The method *checkTravelAlternative* read the selected alternative and send to google maps API a request. The response is a JSON object and through a parser is possible to extract from it the travel object.

View more detailed these steps:

1. There is a control on a travel type selected from the user and there are two possible cases:
   1. ONLYOWNCAR or ONLYPUBLICTRANSPORT or GREEN: when one these travel type is selected, it is created a new travel alternative based on related travel mode (DRIVING for ONLYOWNCAR, TRANSIT for ONLYPUBLICTRANSPORT and BICYCLING for GREEN).
   2. FASTER or CHEAPER: in this case, it is created an array list that contains all possible travel computable.
2. Then it is selected the travel alternative:
   1. In case of cheaper or faster travel, it is necessary to find among all computed travels the cheaper or the faster (are used the two available methods computeTravelTime and computeTravelCost).
   2. In other cases, return the only one alternative computed.

To compute one travel, it is necessary as first step to call the method mapsQuery and pass as parameters the departure place, the destination place, the travel mode (enumeration) and one flag that allow the compute of possible alternatives (true allows alternatives, false no).

mapsQuery performs an HTTP request and must be expressed in the following form:

https://maps.googleapis.com/maps/api/directions/json?origin=StartingPoint&destination=DestinationPoint&mode=TravelMode

It returns one JSONObject that it is directly passed as parameter for travelParser method. This allow to create one Java object from one JSON object, more precisely the travel object.

The method travelParse is able to manage response both the alternative aren’t allowed and otherwise.

public Travel checkTravelAlternative **(**Travel travel**,** TravelType travelType**)** **{**

Travel travelAlternative**;**

**switch(**travelType**)** **{**

**case** ONLYOWNCAR**:**

travelAlternative **=** travelParser**(**mapsQuery**(**travel**.**departure**.**toString**(),** travel**.**destination**.**toString**(),** TravelMode**.**DRIVING**,** **false));**

**break;**

**case** ONLYPUBLICTRANSPORT**:**

travelAlternative **=** travelParser**(**mapsQuery**(**travel**.**departure**.**toString**(),** travel**.**destination**.**toString**(),** TravelMode**.**TRANSIT**,** **false));**

**break;**

**case** GREEN**:**

travelAlternative **=** travelParser**(**mapsQuery**(**travel**.**departure**.**toString**(),** travel**.**destination**.**toString**(),** TravelMode**.**BICYCLING**,** **false));**

**break;**

**case** FASTER**:**

**case** CHEAPER**:**

ArrayList**<**Travel**>** travelAlternatives**;**

**for(**TravelMode travelMode**:** TravelMode**.**values**())**

travelAlternatives**.**add**(**

**(**mapsQuery**(**travel**.**departure**.**toString**(),** travel**.**destination**.**toString**(),** travelMode**,** **true)));**

**}**

**if(**travelType**.**equals**(**TravelType**.**FASTER**))** **{**

Travel fasterTravel **=** travelAlternatives**.**get**(**0**);**

float fasterTime **=** fasterTravel**.**computeTravelTime**();**

**for(**Travel travel**:** travelAlternatives**)**

**if(**travel**.**computeTravelTime**()** **<** fasterTime**)**

fasterTravel **=** travel**;**

**return** fasterTravel**;**

**}**

**else**

**if(**travelType**.**equals**(**TravelType**.**CHEAPER**))** **{**

Travel cheaperTravel **=** travelAlternatives**.**get**(**0**);**

float cheaperCost **=** cheaperTravel**.**computeTravelCost**();**

**for(**Travel travel**:** travelAlternatives**)**

**if(**travel**.**computeTravelCost**())** **<** cheaperCost**)**

cheaperTravel **=** travel**;**

**return** cheaperTravel**;**

**}**

**else**

**return** travelAlternative**;**

**}**