**Algorithm Design**

**3.0**

In this paragraph we will discuss the algorithm design with focus on the interpretation of the calendar on the events inserted by the user and the calculation of the desired transport solutions between two subsequent event by the user preferences.  
To achieve our goal the algorithm that we will design must be divided on two main algorithms:

* The first algorithm will receive the calendar, with the events state of a particular instant, it will recognize the first event and the possible connection between two events (it will recognize a connection between two events in case there’s no events between them, we will focus later on more particular case of overlapping and the end event case), after a connection is recognized the algorithm will call the second one passing to it the two events of which the transport solutions must be calculated. This operation will be repeated until all the events and their connection have been handled.
* The second algorithm, that is in charge to calculate the transport solutions, will receive two events and will first of all consider the user preferences and means of transport that he has registered (such as private means, subscriptions to public services or car/bike sharing services). It will identify a solution from three possible policies: the fastest, the cheapest and the most ecological. For the solution the algorithm will identify a mean of transport by the user preferences (the policy is included in the preferences as well) and a starting and location for the usage of this mean of transport, then it will call the Google external API passing to it the mean of transport and the desired ending location. After this the algorithm will have a portion of the trip covered by the desired mean of transport, until there’s no portion uncovered the algorithm will call itself recursively.

**3.1.0 First Algorithm**

The first algorithm is designed to have a correct behavior on different condition:

* The transport solution for all the event must be calculated:
  + The user changes his preferences;
  + The user register new means of transport;
  + Extraordinary situation: such as loss of data on transport solution;
* An event has been inserted or modified;
* An event has been deleted;

**3.1.1 Formalization of the First Algorithm**

**Input:** Ordered List of Events

**Description**

The event inserted in the calendar can be modeled in an oriented graph as vertex and the necessity of transport solution between events are edges.

So we have a *Graph* as data structure:

V as set of *Vertex*;

E as set of *Edges*;

The object “Event” bundled in a vertex have three attributes: Location, start time, ending time and a Boolean for “ending event”.

We assume that the calendar is passed to the algorithm as an ordered list of events (by the starting time) and we use an ordered list as REP for the set V.

Set E is defined as:

So the object bundled in an edge is a simple pair of events.

We use an ordered list as well as REP for the set E, the order is determined by the event from which the edge is outgoing. If two edges share the same starting vertex the order is determined by the events in which the edges is ingoing. The REP must not contain duplicate elements.

First of all the algorithm fill the set V with all the events received as argument, then insert in the list after the events marked as “end event” a new vertex that have as location the default location.

The algorithm scans the list and add to E an edge between the subsequent events by the ordered lists skipping the events marked as “ending”. Then from each vertex v1 the algorithm scans the edges that is outgoing from it and, (condition c1) if the starting time of the vertex v2 in which the edge is ingoing is before the ending time of v1, for each edge e1 that is outgoing from v2 and ingoing in v3 an edge e2 outgoing from v1 and ingoing in v3 is added. Until there’s no more edges that satisfies condition c1 the operation is repeated. This step is important to grant connection between events also in condition of overlapping.

In the final step the second algorithm will be called for each edge which is contained in E.

This algorithm will be called with specific inputs for any of the situation listed above in **3.1.0**:

* Whole calendar is passed to the algorithm;
* A “small calendar” including the event added or modified and all events “connected” to it as described in the algorithm logic on both sides (antecedent events and subsequent events)
* A “small calendar” including the first subsequent event and all subsequent events “connected” to it as described in the algorithm logic

**Pseudocode**

**//Java pseudocode**

//argument Calendar as List<Event> input

//Edge is a simple class that contains a pair of events and have two methods getFirst() and //getSecond()

List<Event> V;

List<Edge> E;

V.addAll(input);

for( i = 0; i < V.size(); i++ ){

if(V.get(i).isEndEvent()){  
 V.add(new HomeEvent(), i);

i++;

}

else

E.add(V.get(i), V.get(i+1));

}

connectionByOverlapping(E);

for(Edge edge: E)

algorithm2(edge);

connectionbyOverlapping(List<Edge> E){

for(Edge edge1: E){

if(edge1.getFirst().getEnd() > edge1.getSecond().getStart){

for(Edge edge2: E){

if(edge2.getFirst().equals(edge1.getSecond())

addOrdered(new Edge(edge1.getFirst(), edge2.getSecond()));

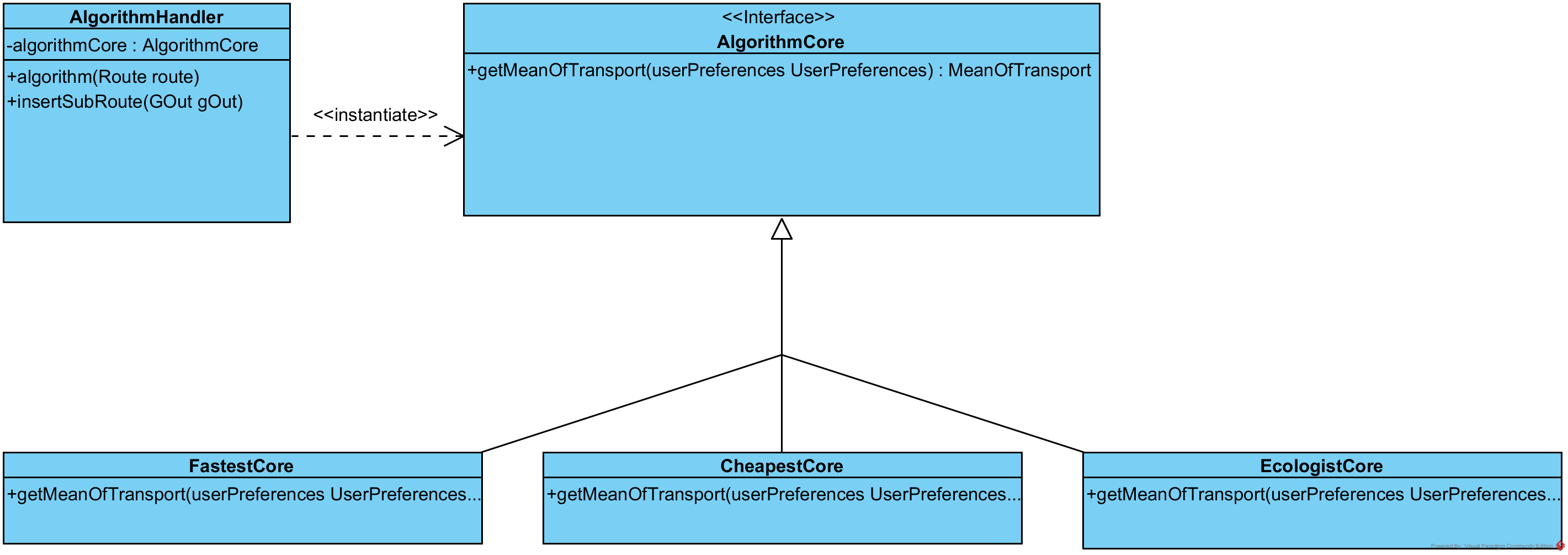
}

}

}

**3.2.0 Second Algorithm**

The second algorithm will receive first of all the two location (l0, l1) and the user preferences which contain the preference on the type of solution Fastest, Cheapest or Ecologist. By this preference the algorithm will choose, similar to a strategy pattern, how to compose his core.



After this first fundamental step the algorithm identifies a mean of transport by the user preferences.   
If this mean of transport is contained in his private mean of transport and is available for the specific situation the Google API will be directly called. If the output of the google API cover all the trip between the two location with the desired mean the algorithm has ended his task, in case not the algorithm call itself with the starting location, as where the Google API indicates that the desired mean of transport will not be used, and with ending location the same ending location which was initially passed to the algorithm (l1). Then he consider as the google solution only for the subroute where the desired mean is used.  
In case the mean of transport desired to use is a public or a sharing one the specific External API will be called and a location is returned (of a bus stop or of the nearest car to be reserved) and used as a location l2.  
First of all the algorithm recall itself passing to it two location (l0, l2). Then call the Google API with the mean (in this case a sharing/public one, but at this level is not important) and follow the same behavior after the Google API call as described above.

In every call of the algorithm at the end the solution returned from the Google API is inserted (entirely or partially depending on the case described above) by a specific function to compose the solution between two events.

**3.2.1**

**Input:** A route

We have route that is a pair of location.

**Pseudocode**

mean = chooseMean(); //call the core as described in the UML in 3.2.0

if(!isPrivateAvailable(mean)){

l2 = callExternalAPI(); //obtain the location from a specific external API

thisAlgo(new Route(route.getStart(), l2));

route = new Route(l2, route.getEnd());

}

gOut = callGoogleAPI(mean, route);

if(!isCoverTotallyByTheDesiredMean(route, mean, gOut)){

thisAlgo(new Route(route.getStart(), whereStopUsingMean(mean, gOut)));

gOut = cutTrip(gOut);

}

AlgorithmHandler.InsertSubTrip(gOut);