Chapter 1

Algorithm Design

In this chapter the most interesting algorithm of myTaxiService are presented using pseudo-code (every developer can easily translate the pseudo-code into the desired programming language). In addition this pseudo-code is referred to an object-oriented programming language, like C++ or Java (the reader can think about the equivalent algorithm in non object-oriented programming language like C, so he has to create manually all the object data structures, the objects himself and he has to define a way to manage and save all the created objects).
The meaning of word *interesting*, used above to define the algorithms which we will present in this chapter, is the following: a characteristic and unique algorithm, used to implement a specific functionalities of this system. For instance, an algorithm to manage the backup of the database can be very complicated in describing the policies, the exceptions and the situations when execute its, but it is a common algorithm for all the systems that store data into a database.

In this chapter the algorithms analysed are the following:

- the city's areas creations and management;
- the queue's management;
- all the algorithms used to handle the special situation which occur when no cabmen are available in the area where the ride (which the Ride Allocator is now assigning) starting position is.

1.1 Map and Areas' Creation Algorithms

Premise. The system has, at its disposal, an XML document that describes the city and its streets. The structure of the document is shown below and it is the following: first there are the four extreme coordinates in order to create a rectangle which contains all the city's area (if the borders of the city are irregular the rectangle's area is bigger than the city's area), then all the streets are listed with the name and two coordinates. In fact, with this two points the street can be represent as the line that joins these two points.

```
<City>
    <Extreme Coordinates>
        <WestX> . . . </WestX>
        <SouthY> . . . </SouthY>
        <EastX> . . . </EastX>
        <NorthY> . . . </NorthY>
    </Extreme Coordinates>
    <Streets>
        <Street direction = "H">
            <Name> . . . </Name>
            <Left Position>
                <X> . . . </X>
                <Y> . . . </Y>
            </Left Position>
            <Right Position>
                <X> . . . </X>
                <Y> . . . </Y>
            </Right Position>
        </Street>
        [. . . ]
        <Street> [. . . ] </Street>
    </Streets>
</City>
```

Figure 1.1: Structure of XML document that describes the city.

For each street an attribute direction is defined: the value H indicates a horizontal road so the two coordinates are the left and the right, while the value V indicates a vertical road so the two coordinates are the high and the low.

When the system starts the map creation it first creates the areas objects. The idea is simple: starting form the Northwest angle of the map it create square areas of side 1.5 kilometres[1].

```
MapCreator.createMap(WestX , SouthY , EastX , NorthY);
horizontalSectors = (EastX - WestX) / 1.5 + 1;
verticalSectors = (NorthY - SouthY) / 1.5 + 1;
for i = 0 -> verticalSectors then
        for j = 0 -> horizontalSectors then
                MapCreator.createArea(j , i);
```

We point out two aspect of the code shown above: first, in the calculus of the number of areas, both in vertical and horizontal, we sum one at the result to count the final area which has a size less than the fixed dimension (1.5 km); second, the notation used in the *for* indicates a cycle of n interactions where n is equal to the value written at the right of the edge.

Now, the objects of type *area* are created, but they do not contain any street. The algorithm used to add the streets is simple: for each street into the XML document (we suppose we have a parser which gives all the streets found into the document as an object), the belonging area is the one where the first coordinates is in. To avoid strange situations where a street is assigned to an area even if only a small part belongs to the area[2] an additional parameter *CORRECTOR* is defined. The parameter assumes on value into the interval [0,1] (the value we have chosen is 0.7). To assign a horizontal street (for the vertical ones is similar)

[1]Due to the city is not perfectly rectangular, this algorithm can create some areas that cover lands out of the city borders. This is not a relevant problem in memory usage.

[2]This situation happens, for instance, when the left coordinate is near the right bound of the area and it has an horizontal direction.

the rules are the following (the map can consider as a grid):

- the row into the map is exactly the one where the left coordinate is in;
- the column into the map is exactly the one where the left coordinate is in if and only if its position is not near the right bound (on the other hand the street is assigned to the next area on the right). Called *size* the dimension of an area and x the distance between the coordinate and the starting of the area *start*, the coordinate is near the area's right bound if:

$$x > start + size * CORRECTOR;$$

Now the algorithm is shown by restricting the use of chain invocations in order to make the algorithm easy to read.

```
//We suppose we have an iterator between the streets, given by the
parser. From now we'll call it parserIt.

  while (parserIt.hasNext()) then
        street = parserIt.next();
        x = street.firstCoordinate().getX();
        y = street.firstCoordinate().getY();
        row = (x - WestX) / 1.5;
        col = (NorthY - y) / 1.5;
        if ( street.type().equalTo('V') && checkY(y , row)) then
              row = row + 1;
        else if ( street.type().equalTo('H') && checkX(x, col)) then
              col = col + 1;
        Map.getArea(row,col).addStreet(street);
```

Now the two methods checkY and checkX are shown.
Both the methods have a boolean return type and try to verify it is needed to increase by one the calculated area (see above for the reasons). The parameter are different according to the type of the street: for a vertical road are required the y-part of the first coordinate and the calculated row while for a horizontal road the x-part of the first coordinate and the calculated column.

```
boolean checkY (double y , int row) {
        start = NorthY - row * 1,5;
        if (y >(start + 1,5 * CORRECTOR)) then
                return true;
        return false;
}

boolean checkX (double x , int col) {
        start = EastX + col * 1,5;
        if (x >(start + 1,5 * CORRECTOR)) then
                return true;
        return false;
}
```

This algorithm generates a first division of the city and its streets into the areas. The algorithm is not perfect and does not consider same particular situations as restricted-traffic zones or busy roads. To improve the quality of the city areas the administrators are able to move some streets between two areas. The administrators are supposed to decide to change a street with some criteria, so no checks are performed on those action.
Finally, the final version of the city is saved on the database, even if a representation is maintained into the Ride Allocator (to be precisely the constructed version).

1.2 Queue Creator Algorithms

The Queue Creator is a subcomponent of the Ride Allocator[3].
The queues' creation is an iterative process performed at the Ride Allocator creation and initialization. After the definition of the map and its areas, the Queue Creator is involved to create one queue into each area.

[3]see the **??** for a complete description.

```
forall Area a in Map then
    a.createQueue();
```

The *createQueue* method is defined into the class Area: it is able to create one object *queue* if and only if it has never created another queue yet. This definition does not require a Factory Method[4] pattern, because exists only one type of queue and no other types can be designed in future. In addition, an exception will be thrown if the method is called when a queue already exists[5].

Into the database this method has no effects because no driver is waiting at the creation of the queues that are empty.

1.3 Queue Manager Algorithms

The Queue Manager has two main algorithms: one to add a driver to a queue by a position and the other one to update a queue (moving down or remove a driver).
When a driver needs to be added to a queue, first the Queue Manager uses the Google Maps APIs to find the street by the position, then identify the area by a method into the Map component. (not shown here). At last is able to add the driver to the correct queue in the last position.

```
void addDriver (Driver d , Position pos) {
        street = GoogleMaps.getStreetByPosition(pos);
        area = Map.getArea (street);
        area.enqueue(d);          //The method enqueue does not
involves any methods into the DBMS. The System Controller, when
asks for a driver adding, also call a similar function into the DBMS.
}
```

[4] see the **??** for a definition.

[5] In a non object oriented programming language this error can be notified using a return value of the method createQueue. However this method can be called only by the Queue Creator, that is involved only once.

When a driver has to be removed from the queue, the Queue Manager gives a specific method which is so simple: it removes only the first element into the desired area's queue. In addition, the method also gives the possibilities to move the driver from the first position to the last one by a flag. The modification need to be stored into the database, so the Ride Allocator, at his return, passes to the System Controller the results of the operations and the DBMS is able to correctly update the queue information. Hence, the Queue Manager only updates the queues, but it does not save any data.

```
void moveDriver (Area a , boolean moveDown) {
        driver = area.removeFirstElement();
        if (moveDown) then
                area.enqueue(driver);
}
```

1.4 Ride Assignment Algorithm

The ride are assigned by the Ride Allocator and inside that, by the Allocator subcomponent. The allocator can access to the Map and, as consequence, to the areas and to the related queues. When the System Controller involves the Ride Allocator to assign a driver to an imminent ride, it delegates also the possibilities to pass through the Client and Users Handler in order to communicate with the drivers, then it waits for the results (they will be saved on the database).
The method used to assign a ride has only two parameters, the starting and the destination position. In an object oriented programming language this method will be defined using the overload principle to allows the possibility to use as starting position either a position or an address. For the destination position the method can accept only addresses. In this description we suppose that the

method receives only addresses[6].
The first passage for the algorithm is to find the correct area associated to the address and in this case the methods of the Map are used, as in section 1.3. Afterwards, the Ride Allocator calls the first driver in the queue and waits for his answer one minute.[7] Here, there are two possibilities: first the driver accepts, so the Ride Allocator immediately removes the driver from the queue and returns to the System Controller; else the driver denies, so the Ride Allocator move the driver to the bottom of the queue and then asks to the next driver into the queue. In both the possible cases the Ride Allocator has a list (the real implementation of this list is not given here) where save the key information about the ride that it is now assigning: so the detected area and the sequence of driver involved. The answers of each driver are not needed because only the last driver in the list has accepted the ride.

```
list assignRide (Address start, Address destination, linkTo-
ClientAndUsersHandler handler) {
        area = Map.getArea(start);
        listToReturn.add(area);
        do
            driver = area.removeFirstElement();
            area.enqueue(driver);
            //Above the driver is re-added to the queue
            listToReturn.add(driver);
            accepted = handler.ask(destination, timer value);
        while (accepted == false);
        area.removeLastDriver();
    }
```

[6]in the version with position, the reader can see the section 1.3 to have an idea about the "conversion"from a position to an address.

[7]In the RASD document the driver has been supposed to answer immediately to a call, but to avoid some infinite waiting of the system (for instance for connecting problems on the driver's device) we define now a timer after which the cabman's answer is consider to be a deny.

Observation: the method ask of the handler asks to a cabman if he want to drive in a ride to a certain destination. The second parameter is the maximum time that the driver has to answer: if the timer expires the method return false (the notification of this event to the cabman is then handled by the Client and Users Handler).

1.5 Special Algorithms

In this section will be given only an accurate description of the algorithms used to administrate the special situation that occur when in some area there is no driver into the queue.
In the previous section, the algorithm that assign a ride does not consider this possibility. In some objected oriented languages this can be accepted and the special algorithm is invoked by an exception. In alternative is possible define a more exhaustive control flow that call the special algorithms into an if condition.
The system does not allow an human-defined sequence of near areas to be called, so in this situation the first area chosen to find a driver will be randomly selected among the areas closest to the one where the starting position is in.
After the choice of the next area the algorithm shown in section 1.4 is called to find a driver for the ride. Obviously, if also this area has no available driver, a new area will be chosen to search a driver who is waiting.

1.6 Conclusion

In this chapter all the algorithms have been presented without consider the possibility to receive more than one rides to be assigned at the same time. If the two or more rides are in different areas, no problem occurs even if the algorithms are called in parallel.
A precisation about the case that happen when two contemporaneous rides start from the same area is required. Calling in parallel the shown algorithm to assign the rides may lead to inconsistent and undesired situations. This prob-

lem can be easily solved by the introduction of synchronization strategies. The implementation of this strategies will not be discussed here, but to have an idea a possible (partial) solution is to have more than one thread to execute ride assignment. When the Ride Allocator is waiting for the driver's answer, it can accepts new requests on an other thread and, if the same area is involved the following driver in the queue will be involved, and so on. Note that this description does not describes all the aspects concerning the synchronization: for instance what happen if the first thread need another driver at the same time of the second one? In addition, if there is only one driver into the queue and he rejects the first ride, the system has to asks to him before enter into the special mode (this is a very strange and rarely case).