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 no 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 do
        for j = 0 -> horizontalSectors do
                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()) do
        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)) do
            row = row + 1;
        else if (checkX(x, col)) do
            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)) do
                return true;
        return false;
}

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

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.

```
forall Area a in Map do
        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].

[3] see the **??** for a complete description.
[4] see the **??** for a definition.
[5] In a non object oriented programming language this error can be notified using a return

1.3 Queue Manager Algorithms

TO BE FILLED

1.4 Ride Assignment Algorithm

TO BE FILLED

1.5 Special Algorithms

TO BE FILLED

value of the method createQueue. However this method can be called only by the Queue Creator, that is involved only once.