OOSWD Project Supporting Document

The following documents is intended as a breakdown of my programme. The programme is a journey and fuel management system for an Irish airline. This document will be broken into four sections Function Specification, Design, Testing and Notes.

Function Specification:

The purpose of the programme is to calculate the most economic route to reach four destination airports. To determine the most economic route, the programme considers the sequence in which airports should be visited and the price of fuel in each of these airports.

The programme is operated from the command line. It takes several inputs from the user. These are:

1. The type of aircraft to be used
2. The home, or starting airport
3. 2 – 4 destination airports, this number is up to the user.

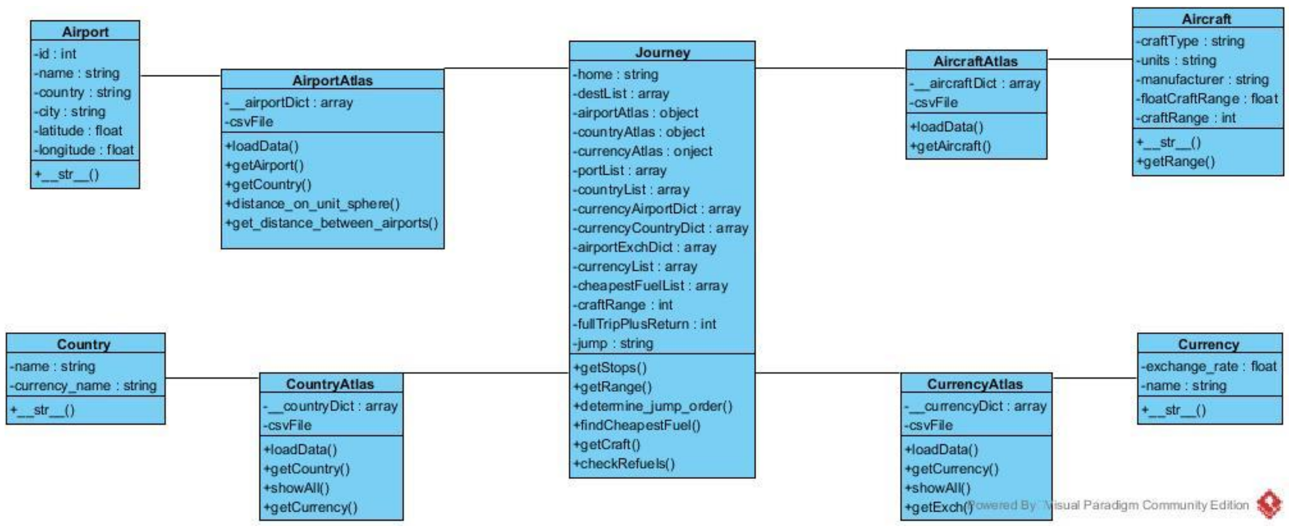
The list of outputs are as follows:

1. The range of the selected aircraft
2. The suggested, most economic route to take between the selected airports
3. The total journey distance
4. The number of times the aircraft will need to be refuelled
5. The airport, or airports, with the cheapest fuel
6. The price of fuel at each of the selected airports
7. If the desired journey is impossible due to insufficient aircraft range

I have not left out any features from the handout, however the programme does not have a GUI, rather it is operated from the command line.

Design:

The below class diagram represents all the classes used in this programme and included attributes and operations/methods.

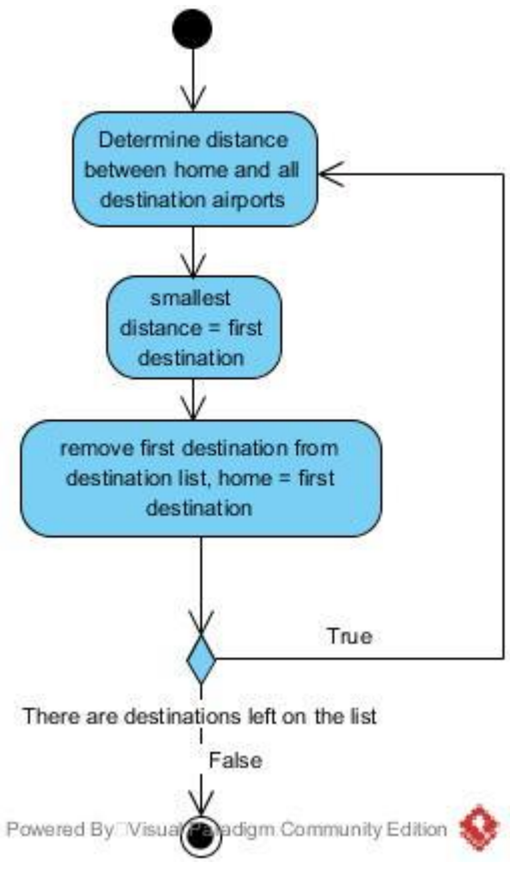


The main class is the Journey, which holds all information the user requires such as aircraft being used, airports to be visited, the sequence in which airports should be visited and the prices of fuel at all the airports that will be visited. The atlas classes take csv files as inputs and populate dictionary objects. These dictionaries are accessed by the Journey class to derive information about aircrafts, airports and currency rates.

<u>Algorithms:</u>

Some of the major algorithms used by the programme are outlined below in flow diagrams.

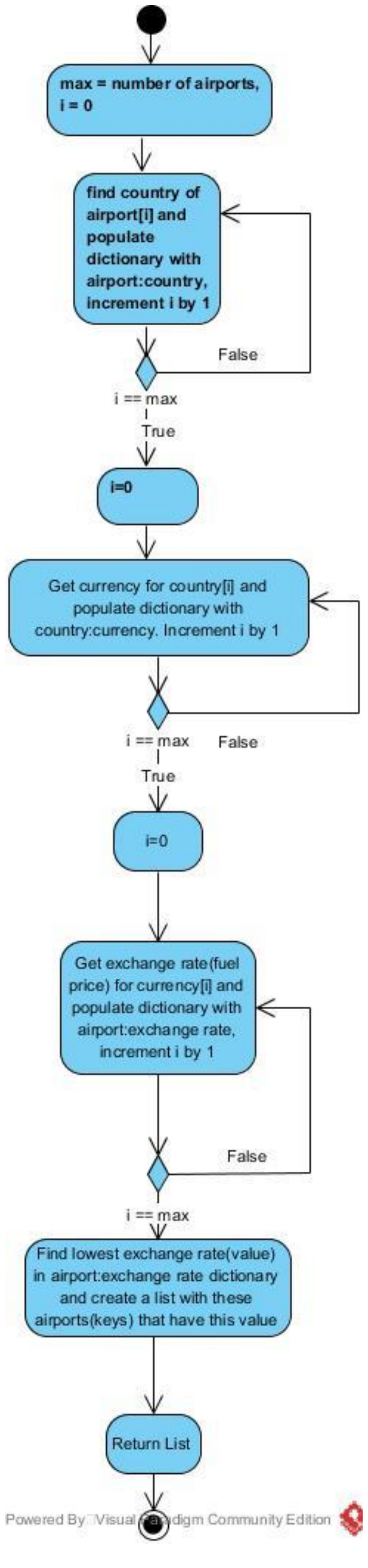
<u>Determine most economic sequence of airports</u>


Determine distance between home and all destination airports
smallest distance = first destination
remove first destination from destination list, home = first destination
True
There are destinations left on the list
False
Powered By Visual Paradigm Community Edition

The diagram above represents the algorithm for finding the most economical sequence in which to visit airports. It takes as inputs a home airport and a list of destination airports. The distance between airports is calculated by a method of the AirportAtlas class. The algorithm calculates the shortest distance in the following way:

1. The home airport is provided by the user
2. Several destination airports are provided by the user
3. The programme calculates the distance between the home airport, and all the other airports.
4. The closest airport to home is becomes the first destination
5. The algorithm then repeats itself, out of the remaining destinations, the closest one to destination 1 is found and becomes destination 2.
6. This repeats until there are no airports left.
7. The total journey distance is calculated as the sum of the trips taken between airports, + the distance between the original home airport and the last destination airport (the return trip).

Find airport with cheapest fuel (next page)


max = number of airports, i = 0
find country of airport[i] and populate dictionary with airport:country, increment i by 1
False
i == max
True
i=0
Get currency for country[i] and populate dictionary with country:currency. Increment i by 1
i == max
False
True
i=0
Get exchange rate(fuel price) for currency[i] and populate dictionary with airport:exchange rate, increment i by 1
False
i == max
Find lowest exchange rate(value) in airport:exchange rate dictionary and create a list with these airports(keys) that have this value
Return List

The above diagram represents the algorithm to determine which airport/airports, has/have the cheapest fuel. Fuel price is determined based on exchange rate. The algorithm makes use of several lists and dictionaries. It makes use of information stored in atlas objects.

<u>Error handing:</u>

The programme has extensive error handing implemented. Any action that could potentially crash the programme has been predicted and a strategy implemented to prevent it.

Testing:

I have extensively tested this programme using unit tests. I made sure to create a test for every class and that every method, which is used in the normal functioning of the programme, is tested. Some of my methods do not return values, rather they print an output. This presented a problem for my unit testing, as the method I had learned to unit test required a return value to compare to an expected output. I solved this problem by redirecting the contents of stdout to a byte array and then compared the contents of the byte array to my expected outputs. This has been extensively documented in the unit test file.

Notes:

<u>Note1:</u>

AirportAtlas.loadData:

```
def loadData(self, fileName):

    try:

        with open(fileName, errors = 'ignore') as csvfile:

            reader = csv.DictReader(csvfile)

            for row in reader:

                self.__airportDict[row['Code']]= Airport(row['AirportID'], row['AirportName'],
row['Country'], row['CityName'], row['Lat'], row['Long'])

    except FileNotFoundError:

        print("No CSV file found.")
```

This method populates a dictionary object with information derived from a csv file. The programme requires a solution to deal with erroneous data in the csv file. I opted to use “errors = ‘ignore’”, which is a value associated with the ‘open’ function, which tells it how to deal with erroneous data.

There is a good reason why I didn’t use a try catch for this, which would look like this (lines added in red):

```
def loaddata(self, filename):

    try:
        with open(filename) as csvfile:
            reader = csv.DictReader(csvfile)
        try:
                for row in reader:
                        self.__airportDict[row['Code']]= Airport(row['AirportID'],
                row['AirportName'], row['Country'], row['CityName'], row['Lat'], row['Long'])
        except Exception:
                pass
      except FileNotFoundError:
        print("No CSV file found")
```

This does not have the desired effect. What I want to achieve is for the reader to skip any lines it has a problem with, and simply move onto the next line. Using "errors = 'ignore'" has exactly this effect, however the try catch causes the reader to stop reading once it encounters an error. This can be tested easily. If you create an instance of AirportAtlas using the try catch, and you attempt to use the getAirport method, inputting an airport code from near the bottom of the csv file, it will fail. This is not because there is an erroneous line about halfway through the csv file, the try catch causes the programme to stop reading the file at this point. If you repeat the test using the "errors: 'ignore'" method, the getAirport method will return airports from anywhere in the file.

This note applies to all atlas classes that read a csv file including; AirportAtlas, AircraftAtlas, CountryAtlas and CurrencyAtlas.

Note 2:

Inputs for this programme are not case sensitive. Although all aircraft codes and airport codes are upper case, I made sure to convert all inputs to upper case so that it doesn't matter what case the user enters them in. This is achieved in the below code:

```
craft = input("Please specify the code for the aircraft you will be using:")
craft = craft.upper()
```