**Flight Documentation**

The purpose of this documentation is to outline the software development lifecycle from design through implementation and testing of a small Irish software company’s design of a fuel management software solution for a large Irish Airline. What will be covered, will be the details modules, classes and functions used to achieved this, including the .CSV files used to read the data into the program and the data that we will output to a .CSV file as well as the testing strategy used.

The goal of this software solution is to calculate the cheapest possible route with regards to refuelling an aircraft when flying between 5 airports on an itinerary, which will be read from a .CSV file that outlines multiple routes while specifying an aircraft to be used for each route, which will have a limited fuel capacity.

To do this, data files regarding airport information, currency conversion rates, countries currency information, aircraft information will be utilised to:

* Utilise airport geographical locations to calculate distances between airports
* Utilise currency rates to calculate fuel costs at each destination.
* Utilise the distance multiplied by the fuel cost to weight the cost of a route.
* Find the cheapest route by finding the cost of all possible combinations of flights in each route and picking the cheapest combination.

Features implemented will be a Text Based User Interface. This will be the primary way in which the program will be used. This will allow the user to:

* Calculate the cheapest possible Route from itineraries listed in a .CSV and writing the results to a .CSV file. The user can specify what the file input and output will be
* Input a country and see which currency is used or with a given airport or airplane code, the relevant details will be returned.
* Build a custom itinerary in the program by entering airports and an aircraft and having the cheapest route

While calculating the cheapest possible route, the program will determine if any flight within the route exceeds that of the range of the aircraft assigned to said route. If this happens, the program will resume while printing an appropriate error message to the data output (both in console and when writing data output to a .CSV file

While taking in 5 airports, the program will rebuild the list to include the flight back to the beginning airport that must be included.

Small jazz:

* In the text UI when asking for inputs for aircraft, minimised code with efficient if loop within while loop while error handling and appending to list.
* When asking for file to read from or write to, the program ensures that .csv is included in name and that at least 1 letter before it is present.

FEATURE NOT INCLUDE FROM THE SPEC – stopping off at an airport twice. This was almost completed, and may still be completed at the last minute. Same for outputting the file to a specific directory.

Features I would implement

* In the text UI, I would work to make the “loopreturn” feature more modular. This is when the user has selected an option (example is “M” for currency details of a country). When in this option, we call it the “M” loop.

The issue stopping this for now that for the suer to return to the menu, we set the break condition as “loopReturn = False” along with loop\_C = False, which allows the user to return to the main menu from the “M” option.

To do this I would put it in its own function, and make a variable that would take a slice of the letter “M” (this would change for every menu option, example “C”, “A” etc) from the corresponding variable which has been set to “True” to enable the menu loop. This would then be used as a placeholder in the function so whatever calls it would pass in the variable and ultimately break the relevant loop .

* In text UI, when asking for file to read from / write to, add an if statement to determine if last 4 characters are .CSV, if not, then add it.
* A GUI was half finished using Tkinter(). Time was spent familiarising myself with the module and building a few simple functions. (this code will be pasted below) I then went on to create a separate GUI class and was to simple have a few data entry fields and then call a function using event handling. But there were difficulties with linking the separate class file to the main text based UI, meaning this was abandoned due to time constraints

(2) Design: Assumptions, Inputs and Outputs. Describe/Illustrate your object model

and how your program was designed (max. 1 page text and 2 page diagrams)

• Class diagram

• Block diagram of main algorithm(s)

• Text description to explain diagrams

(3) Testing: How have you tested your program. Have you got a suite of tests that

exercise the key components? (max. 0.5 page)-------------

Design: Assumptions, Inputs and Outputs. Describe/Illustrate your object model and how your program was designed (max. 1 page text and 2 page diagrams) Class diagram Block diagram of main algorithm(s) Text description to explain diagrams

**Input**

CSV files:

* “testroutes.csv” where each line represents an itinerary of 5 airports (starting with Home airport) and the Aircraft type for that route. (no header) *E.g.*

*DUB,LHR,SYD,JFK,AAL,777  
SNN,ORK,MAN,CDG,SIN,A330  
BOS,DFW,ORD,SFO,ATL,737  
DUB,ORK,ORD,JFK,AAL,777*

* *“aircraft.csv” where each line* represents an aircraft and its properties. (no header) *E.g*

*A319,jet,metric,Airbus,3750*

*A320,jet,metric,Airbus,12000*

* *“airport.csv”* represents all airports that we will utilse when building an Airport Atlas. (no header) E.g.

*596,Cork,Cork,Ireland,ORK,EICK,51.841269,-8.491111,502,0,E,Europe/Dublin*

*597,Galway,Galway,Ireland,GWY,EICM,53.300175,-8.941592,81,0,E,Europe/Dublin*

*599,Dublin,Dublin,Ireland,DUB,EIDW,53.421333,-6.270075,242,0,E,Europe/Dublin*

* *“countrycurrency.csv”* represents what countries use which currency along with other details pertaining to a country. (no header) E.g.

Ireland,Irlande,IE,IRL,372,IRL,ie,IE,IRL,353,IRL,EI,119,IRL,EUR,IRELAND,2,Euro,978,Yes

Isle of Man,Île de Man,IM,IMN,833, ,uik, ,GBM,44,GBM,IM,120, ,GBP,ISLE OF MAN,2,Pound Sterling,826,Crown dependency of GB

Israel,Israël,IL,ISR,376,ISR,is,IS,IL,972,ISR,IS,121,ISR,ILS,ISRAEL,2,New Israeli Sheqel,376,Yes

* *“currencyrates.csv” lists all currencies, their codes and their exchange rats* (no header) E.g

*Euro,EUR,1,1*

*Finnish Markka,FIM,0.1682,5.9457*

*Falkland Islands Pound,FKP,1.4029,0.7131*

*French Franc,FRF,0.1524,6.5596*

*British Pound,GBP,1.4029,0.713*

*Text based User Interface will take input from the user and present the relevant information. Command will usually be in the form of predefined options (letters representing the option) and string based input when entering countries, airports, and aircrafts.*

**Output**

Calculations for cheapest route will be added to a .CSV file, which will default to "eoinoutput.csv", but there are occasions when the user will choose the output file name.

Information such as results to unit tests, calculations for shortest route, currency/airport/country data will be output to the screen on the python console, via the text based user interface which will be launched via an IDE.

**Assumptions**

1. All figures will be converted to Metric., those aircrafts notes in “aircraft.csv” that stated imperial, have had their range converted to KM.

2. Route must return to the beginning city.

5. All airports must be visited at least once.

6. Pilot is permitted to re-visit one of the airports ONCE, if it is more efficient to do so. (not incorporated)

7. Currencies rates will be calculated from a static CSV file. (Potential option to use daily, weekly, or live data for calculations)

8. Fuel costs the same in every Country. We are only concerned with the exchange rate.

9. All aircrafts consume the same amount of fuel per KM travelled.

10. It costs €1 per litre of fuel and 1 litre of fuel allows 1 KM of flight.

11. Where countries do not use euro, the cost of fuel will be calculated at the local currencies “toEuro” exchange rate i.e., GBP to EUR will be. GBP1 = e1.4029. 1000 litres of fuel will cost €1402.

12. If the flight cannot make the flight (as it’s range is below the distance of the flight) the appropriate error message should be returned.

13. The distance between airports is calculated as the great circle distance between

Them, using latitude and longitude details input from “airport.csv”

14. Main cost figures are calculated on the basis that the plane fills only the amount required to get to the next airport. Further refinement is separated, with more to follow in the future.

**Algorithm:**

1. Read CSV file of Airport details via **AirportAtlas** class.
2. Pass the data into **Airport** Class to fill corresponding attributes
3. Create a Dictionary of all Airports
4. Use “airportCode” as the dictionary’s key. Corresponding values should be Airport Objects
5. Read CSV file of planned routes. All airports on this route should be visited once, allowing a return to one of the airports twice.
6. Create a variable “routes” which is a list where each item is an airportCode from the planned routes CSV. Also create a variable “aircraft” which takes the corresponding aircraftCode from the planned routes CSV.
7. Make a variable named itinerary, which will be an **Itinerary** Object corresponding to each line of the planned routes CSV. This will be achieved by
   1. passing the list of airportCodes created in step6 to the **Itinerary** Class, where the attribute “routes” will consist of a list of **Airport** Objects (achieved by passing each airportCodes from the list to the **getAirport()** function of the **AirportAtlas** Class) ..
   2. passing the aircraftCode created in step6 to the Itinerary Class, where the attribute “aircraft” turns the aircraftCode into an **Aircraft** Object by calling the **Aircraft** Class and passing the aircraftCode to it as an argument.
8. For each route in the planned routes CSV, an “itinerary” will be created and appended to the “**itineraryList**”
9. Each item in the “**itineraryList**” will be iterated through and have all airports, excluding beginning and end, permutated.
10. All permutations are then rebuilt with their corresponding beginning and end airports attached. Each rebuilt route will then be appended as a list item, to the list “**permutatedItineraryLists**”.
11. All permutations of each route will then be appended as a list item to the list “**permutatedItineraryMotherList**”. *This allows us to distinguish when the permutations of one route end and another begins, as they will be their own list item in “****permutatedItineraryMotherList”*** *.*
12. Each itinerary in the “***permutatedItineraryMotherList”*** *from the above step is now**iterated through, and each airport within the itinerary has the distance calculated from the 1st to the 2nd, the 2nd to the 3rd, and so on.*(this is done for every permutation of every route)
13. The distances are multiplied by the destination’s “toEuro” currency conversion rate to get the cost of that flight.
14. This total cost figure is added for all flights (legs) in the route (journey). The cheapest route for each itinerary is then appended to the list “**listoOfCheapestJourneyRoute**”
15. If a flight (leg) is greater than the aircrafts distance, then it will be recorded that the whole route cannot be completed. The program will not halt.
16. All of the above lists
17. For each route, create an Object Aircraft.

Particularly in the pricing calculations.

How the best price distance / route is calc

How it fits together, with methods being taken from different classes (of if it is all in the one)

**To Build Airports:**

Upon initialisation of the **AirportAtlas *class:***

* We call the **buildAirportDic(“airport.csv”)** method, which returns **airportDic. airportDic** is a*dictionary*of **Airport *objects***, which reads from a CSV file, and calls the **Airport *class*** for each relevant piece of information from the CSV that was read in. The Key to this dictionary is the airportCode. By calling this dictionary and passing in an airport code, an airport object with various corresponding attributes can be accessed.

**To Build Itinerary:**

Upon initialisation of the **BuildItinerary *class:***

* We call **constructItineraryList(AirportAtlas *object*, “testroutes.csv”)**. This returns **itineraryList**, which is a list of itineraries, composed of Itinerary Objects, made by calling the **Itinerary Class**. Itinerary *Objects* consist of a *list* of routes where each airport is an *object,* and aircraft which is also an object.
* We then call **permutateItineraryList()** which will return **permutatedItineraryMotherList,** whichis a list where each item contains lists of all possible permutations of 6 airport *objects* when the 1st and 6th is the home airport.
  + This is done by iterating through the itineraryListandslicing away the home journey at the beginning and end, these middle slices of the journey are then permutated and appended to a list listOfPermutatedMiddleJourneys**,** which is then iterated through so each permutation can have the home airport added to the beginning and end. Each of these rebuilt journeys is then added to the list that is returned, permutatedItineraryMotherList**.**
* We then call **findCheapestDistance** ()which returnslistoOfCheapestJourneyRoute**,** which is of the cheapest possible route for any 6 airports where the first and last is the home airport and every city must be visited at least once.

**To work with correct currency:**

Upon initialisation of the **CountryCurrencyAtlas *class:***

* We call the readCurrencytoList(“currencyrates.csv”) method, which reads the CSV file and returns the *list* currencyRatesList, which contains the relevant information from the CSV.
* We then do the same with the readCountiestoList(“countrycurrency.csv”) method, reads the CSV file and returns the *list* countryCurrencyList which contains the relevant information from the CSV.
* We then create a countryCurrecnyDic *dictionary* by calling the buildCountryCurrencyAtlasDic() method, this compares each item in currencyRatesList which each item in countryCurrencyList, to find which list items share the same currencyCode details. When a match between lists is found, it will build the *dictionary* where the key is the countryName and the value is a **CountryCurrency *object*** made up of the relevant information from both lists that had matching currencyCodes.
* By calling the **getCurrency(country)** method, we access the dictionary using Country as a Key, and have access to all available currency information. This will mainly be done by passing the Airport object’s country attribute to get exactly what currency that airport will be using.

**To find cheapest route of an itinerary of 6 airports with a specific aircraft:**

We will use the **permutatedItineraryMotherList** that was returned from the **permutateItineraryList()** function. First we must:

* Iterate through each item of this *list* of *lists*, where each *item* represents all possible permutations of each itinerary.
* For each of the above iterations (i.e possible permutations of a route) we will find the distance of each leg (i.e airport 1 to airport 2, airport 2 to airport 3 etc…). At this point the distance of each leg will be compared to the range of the aircraft assigned to the route. If the distance is greater than the aircrafts range, then the distance of that leg will be assigned as a “9999999999*9999999999*” thus rendering the entire route unachievable.
  + (Distance will be calculated using the **“**getDistanceBetweenAirports**”** method in the **AirportAtlas** class, which takes the latitude and longitude of 2 airport objects and passes them to the *static* method **“calcDistance”,** which then passes the latitude and longitude theough tro the static method **“definetheta”,** which converts them *to theta and phi* figures, which **“calcDistance”** then uses to complete and return the answer to the formula “distance = (acos(sin(theta1)\*sin(theta2)\*cos(phi1-phi2)+(cos(theta1)\*cos(theta2)))\*radius\_earth)”)
* The distance of each leg is multiplied by the costOfFuel at the destination (achieved by fetching the “.toEuro” conversion rate of the destination country). Thus giving the total costOfEachLeg.
* The costofEachJourney is calculated by accumulating the costOfEachLeg. At this stage, any journey containing a leg greater than that of the aircrafts range will have been calculated with a distance of *99999999999999999999 (important for the next step)*
* We will then find the cheapestJourneyCost. This variable was pre-assigned as 100000000000000 , a number purposefully larger than all our possible routes, yet smaller than the distance assigned if the leg of the journey is not is greater than the aircrafts range. We then check if costofEachJourney is less than cheapestJourneyCost, if it is smaller, then we reassign cheapestJourneyCost as costofEachJourney, and continue to do this for every possible permutation of the route. The cheapest for each route in the itinerary is then assigned to listoOfCheapestJourneyRoute. Any routes that can’t be done due to limited range of the aircraft will be left as default error value *“[""ERROR: This route can't be achieved with this aircraft"", 100000000000000]"* when printing to the screen or CSV file.

**To begin optimisation of fuel purchases:**

* First it is required to rank the exchange rates of each airport of the route, using the **rankExchangeRates()** method which returns **“**list**\_**ofExhangeRankings**”**. This is done by iterating through the ItineraryListand using the **getCurrency()** method to find the “toEuro” exchange rate for each airport in each route. Ultimately, the airport and the exchange rate is appended as a tuple to a list and is simultaneously sorted by exchange rate.
* To add a number to each ranking the method **“addNumberToRanking()”** iterates through **exhangeRateRanking\_List** to further iterate through each airport in ascending order from cheap to expensive, while adding it’s ranking into the tuple. These are appended to a list and returned as **“**numberToRanking\_List**”**
* For the method **“efficientFuelingAtHome()”** we will compare all of the exchange rates of the countries in the itinerary. We will then assess where the home country sits in this ranking. If the starting country (home) rate for fuel is the best value. A full tank will be purchased. If home is the least value, the minimum amount to fly/get to the next destination is purchased. This is achieved with multiple “if” statements, where by if the home county == the first or second position in the list of rankings, then a full tank is purchased. If the position is last, then the minimum is bought, with a gradient for those in between.
* The minimum amount of fuel required to fly (1000 litres) is added as default. If a full tank is to be bought, the initial 1000 is subtracted. If home value is not ranked favourably, only what is required on top of the initial 1000 litres will be bought to get to the next airport. For now the calculation end here, but it will evolve to incorporate the whole journey in time.

We will use the **permutatedItineraryMotherList** that was returned from the

Test Strategies:

Unit testing was employed as a means to test the soundness of various aspects of the system during its creation.

For the most part these tests passed and failed when they were meant to. However there were times that these tests highlighted errors that needed to be fixed with error handling or better structuring of data.

To set up these tests, a new testing class was created called “KnownValues” which imported and utilised the unittest module. The purpose of the “setup():” function was to establish known values where possible (e.g. distances between 2 locations determined via the internet, or what range and manufacturer should be returned for a given Aircraft).

Objects of classes were then instanciated to allow testing and the unittest methods “assertEqual()” and “assertNotEqual()” were used to determine if these values matched as expected. Another tool used was the “assertRaises()” method to determine if error handling acted accordingly.

Some tests performed were on

Aircrafts and the values they should return.

Different Currencies and that their attributes were assigned correctly.

The distances between a number of airports.

Openining missing files and ensuring the relevant error was raised.

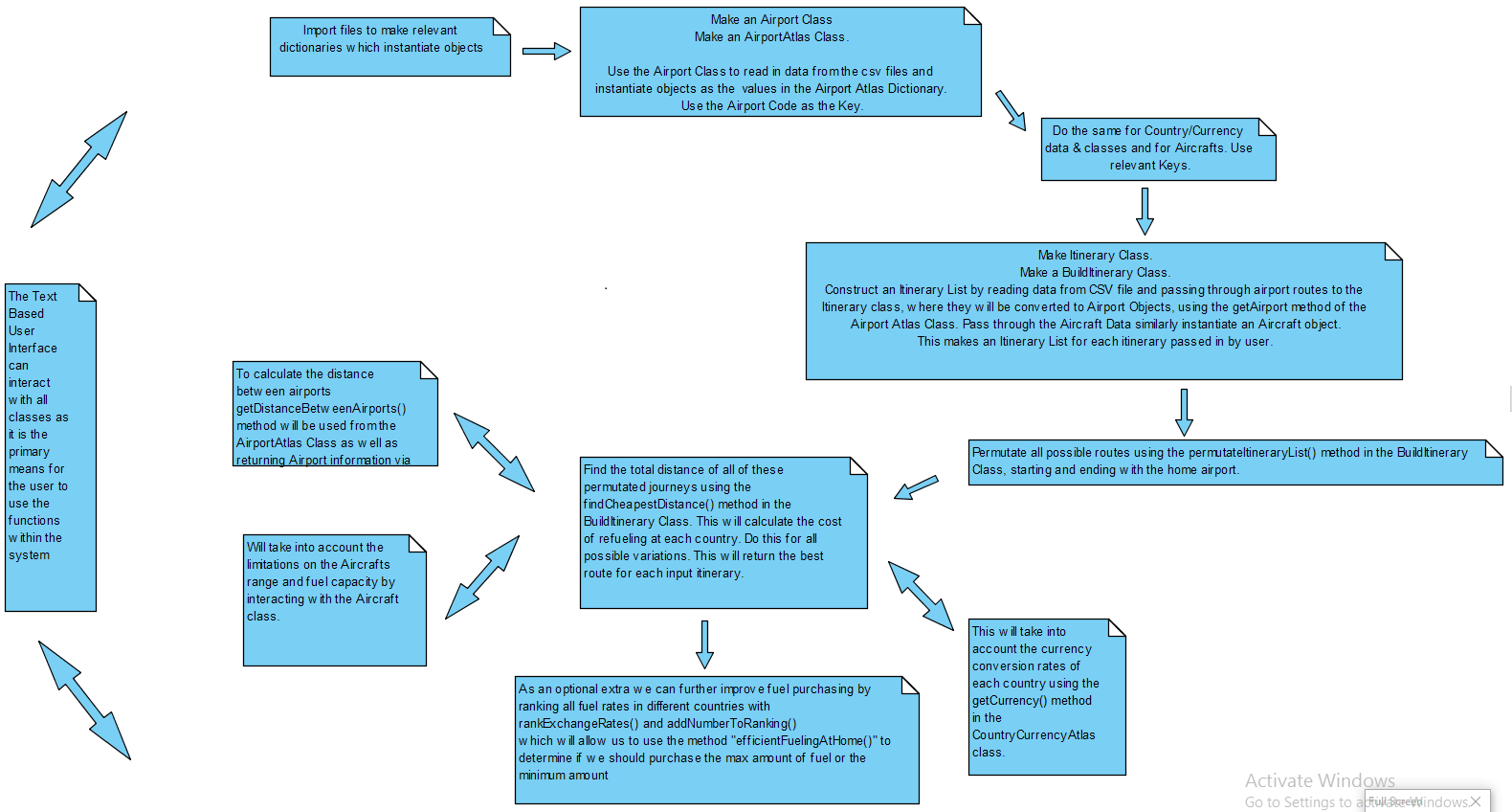
Testing that the correct number of permutations was made per given input.

Additionally a system test was performed that spanned more than one unit to ensure classes communicated sufficiently. This was tested by using the .country attribute after using the “getAirport()” method of the AirportAtlas Class as the paramater“getCurrency()” method of the CountryCurrencyAtlas Class and finally getting the .currencyName attribute from the result.

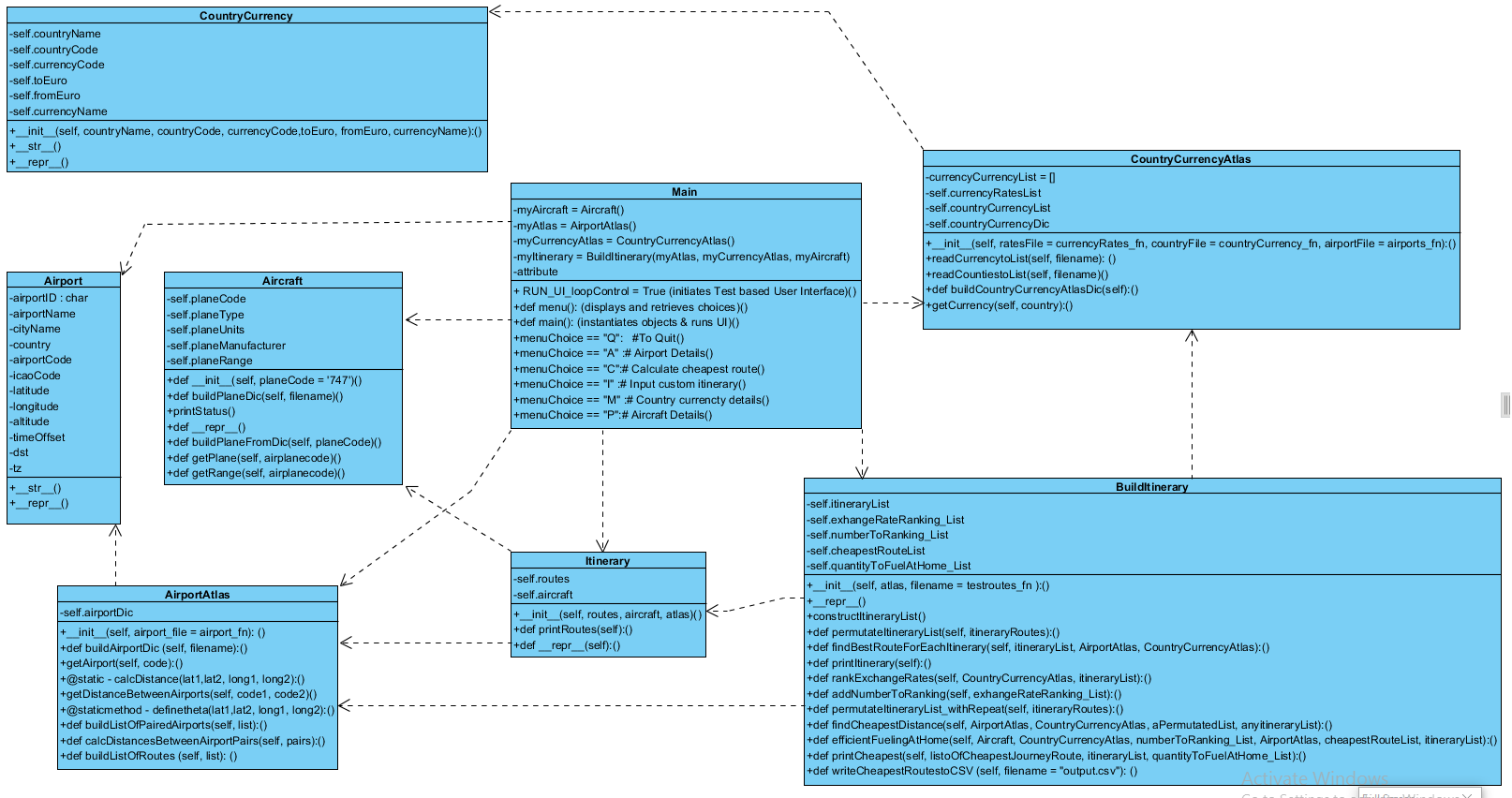
Test\_Airport\_Currency\_DUB\_EURO = self.myCurrencyAtlas.getCurrency(self.myAtlas.getAirport("DUB").country).currencyName

Debugger was used throughout.

Block diagram



Class Diagram



**from** tkinter **import** \*  
clicks = 0  
nightmode = 1  
  
############################################# GUI #############################################  
  
**def myGUI**():  
  
 **def leftClick**(event):  
 print("left")  
  
 **def middleClick**(event):  
 print("middle")  
  
 **def rightClick**(event):  
 print ("right")  
  
  
 **def printName**(event):  
 # b=1  
 # print("button clicked")  
 printscreen.configure(text= "!!TESTING!!")  
  
 **def callback**(whichButton):  
 b=1  
 # print("button clicked")  
 printscreen.configure(text= whichButton + "button clicked")  
  
  
 **def countClick**():  
 **global** clicks  
 clicks +=1  
 clickMeCounter.configure(text = clicks)  
  
 **def nightMode**():  
 **global** nightmode  
 nightmode +=1  
 **if** nightmode % 2 == 0:  
 mainGUI.configure(background = "#7A7A7A")#FFFFFF  
 nightmodeButton.configure(text="---Click for Daymode---")#FFFFFF  
 **else**:  
 mainGUI.configure(background = "#FFFFFF")  
 nightmodeButton.configure(text="---Click for Nightmode---")#FFFFFF  
  
 options = ["calculate distance", "show prices", "show itinerary", "calculate cheapest"]  
  
 mainGUI = Tk()  
  
 # topFrame = Frame(mainGUI)  
 # topFrame.grid(row=0)  
 # bottomFrame = Frame(mainGUI)  
 # bottomFrame.grid(row=30)#(side=BOTTOM)  
  
 mainGUI.title("Eoin Flight GUI")  
 mainGUI.geometry("800x800")  
 # window.wm\_iconbitmap('eoinlogo.ico') ##single quotes?  
  
 picture = PhotoImage(file = "giphy.gif") #must be giff or pgm/ppm unless imaging library used.  
 pic = Label(mainGUI, image = picture)  
 pic.grid(row=0)  
  
 label1 = Label(mainGUI, text="please enter... below")  
 label1.grid(row=1,column=2)  
  
 entry1 = Entry(mainGUI)  
 entry1.grid(row=1, column=3)  
  
 button1 =Button(mainGUI, text="click button for", fg = "red", command=callback)  
 button1.grid(row=2)  
  
  
 clickMeCounter =Button(mainGUI, text="click counter", fg = "blue", command=countClick)  
 clickMeCounter.grid(row=3)#(side=LEFT)  
  
  
 nightmodeButton = Button(mainGUI, text="---Click for Nightmode---", command=nightMode)  
 nightmodeButton.grid(row=4)#(side=RIGHT)  
  
  
 printscreen = Label(mainGUI, text="show here")  
 printscreen.grid(row=5)  
  
  
 checkbox1 = Checkbutton(mainGUI, text = "Do you want.....")  
 checkbox1.grid(row=6, columnspan = 2)  
  
 TESTINGbutton = Button(mainGUI, text = "TESTING")  
 TESTINGbutton.bind("<Button-1>", printName)  
 TESTINGbutton.grid(row=8, column=3)  
  
 airport1\_label = Label(mainGUI, text="please enter home airport")  
 airport2\_label = Label(mainGUI, text="please enter 2nd airport")  
 airport3\_label = Label(mainGUI, text="please enter 3rd airport")  
 airport4\_label = Label(mainGUI, text="please enter 4th airport")  
 airport5\_label = Label(mainGUI, text="please enter 5th airport")  
 airplane\_label = Label(mainGUI, text="please enter aircraft")  
 airport1\_entry = Entry(mainGUI)  
 airport2\_entry = Entry(mainGUI)  
 airport3\_entry = Entry(mainGUI)  
 airport4\_entry = Entry(mainGUI)  
 airport5\_entry = Entry(mainGUI)  
 airplane\_entry = Entry(mainGUI)  
  
 airport1\_label.grid(row=15, sticky=E)  
 airport2\_label.grid(row=16, sticky=E)  
 airport3\_label.grid(row=17, sticky=E)  
 airport4\_label.grid(row=18, sticky=E)  
 airport5\_label.grid(row=19, sticky=E)  
 airplane\_label.grid(row=20, sticky=E)  
 airport1\_entry.grid(row=15, column=1)  
 airport2\_entry.grid(row=16, column=1)  
 airport3\_entry.grid(row=17, column=1)  
 airport4\_entry.grid(row=18, column=1)  
 airport5\_entry.grid(row=19, column=1)  
 airplane\_entry.grid(row=20, column=1)  
  
  
 counter = 0  
 **for** i **in** options:  
 # counter = 0  
 button2 = Button(mainGUI, text="click for " + i, command = **lambda** buttonTitle=i: callback(buttonTitle))  
 # button2.pack(side=tkinter.LEFT)  
 button2.grid(row = (9+counter))#(fill=X)  
 counter +=1  
  
  
 mainGUI.mainloop()  
####################################################################################################################################################################################