Object Orientated Software Development I
Application
Plane Sailing Route Planner

Jamie Coyle
C11431308

December 18, 2015

Function Specification

For this application I have tried to create a program to allow a user to achieve multiple things. The program is able to return details of an airport, given the airport's 3 letter code, return the details of any plane that is available for Plane Sailing to utilise and return the currency details of a given country. The program will give the option of whether one would like to calculate the shortest or cheapest option for a given route. Routes may be loaded into the program in one of two ways, through a csv file which must be in the same directory as the program, or manually loaded one by one. If manually loaded, a specific aircraft must be designated. If the routes are loaded via a csv file, then the output will be printed to the command line, as well as being outputted to a csv file, with the name being chosen by the user.
Interaction with the application is mainly through a command line interface, but there is the option to manually input a route through a graphical user interface. When calculating the shortest and cheapest distance, if the designated aircraft is not able to complete the journey due to its range, then an error informing the user of this is printed in all the relevant places. An alternate route is not found by the program. The program does not consider any optional layover stops to reduce cost.
Something to look out for is that I attempted to convert the *aircraft.csv* file to a *.json* file to attempt to save memory in the system. Doing it in this way eliminates the need for the program to store the aircraft details in a dictionary within the system. My program creates the file, searches through it as needed, and deletes it upon termination of the application.
Assumptions:

- All files, including custom itineraries, are located within the same directory.
- When manually inputting airports, they must be entered one at a time.
- The program assumes any input file used to pass in multiple routes at once, is a *.csv* file. It will have only 5 airports on a specific route and they

will already have the correct code in it. Each route will have a correct plane associated with it.

- Interaction with the program is achieved only through the ***UI.py*** file.

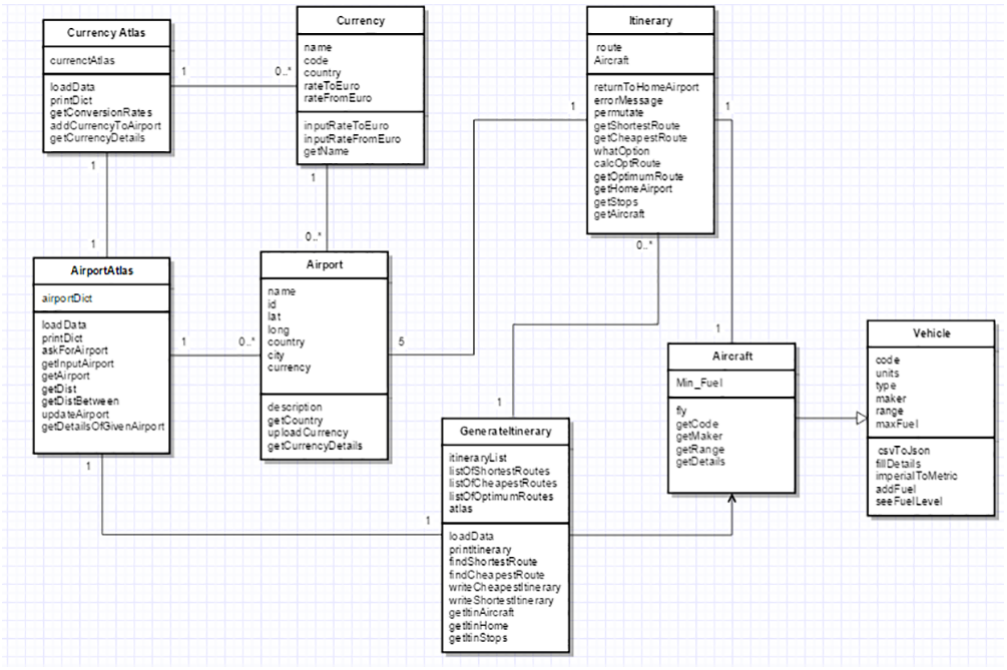
Inputs:

- aircraft.csv
- countrycurrency.csv
- currencyrates.csv
- airport.csv
- plane.ico (not fully needed)

Outputs:

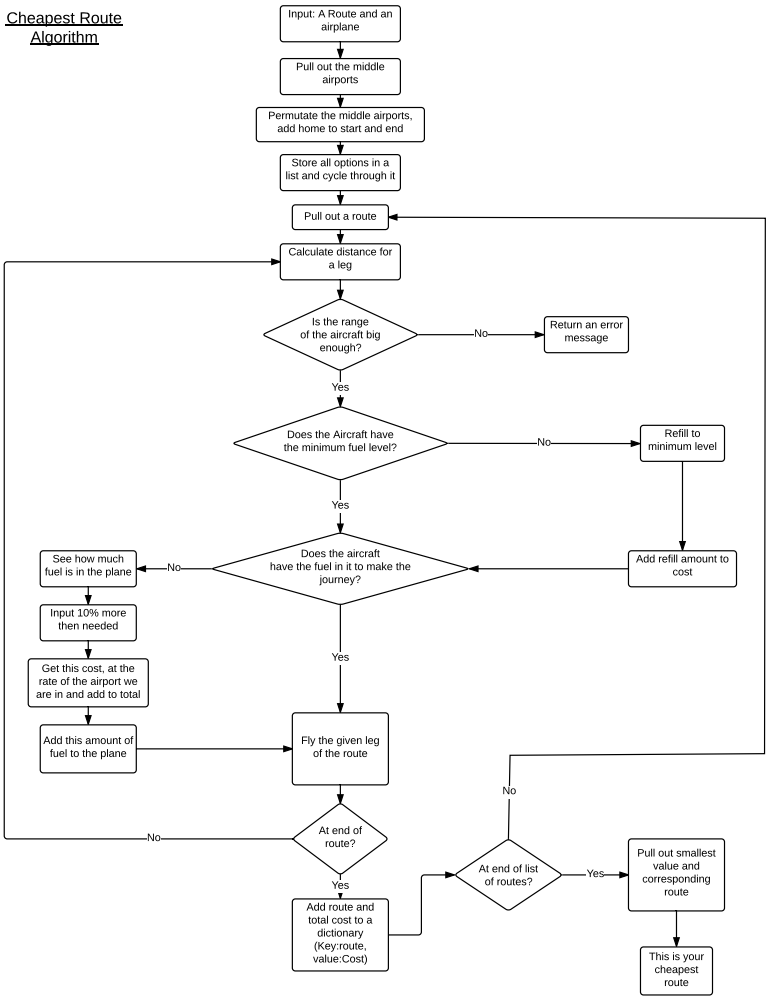
- If a file input was selected, a csv file detailing the best route options.
- A route, detailing what airports to visit in what order, coupled with either a cost in Euro or a distance in KM.

Design



The above figure outlines how I designed the classes for this project. Lines in the diagram represent relationships between classes. Numbers at the end of the lines indicate how many objects participate in a specific relationship. An empty

arrow head indicates inheritance. A filled arrowhead indicates a uni-directional relationship, all other arrowheads are bi-directional relationships.
A few things to note. Each Airport object has a Currency attribute, and this hold the specific Currency object associated with that airport. An Itinerary object is made up of a list of 5 airport object and an Aircraft object. The Aircraft object inherits from the Vehicle class. This is to accommodate the expansion of other transport ventures made by Plane Sailing. Some modifications will need to be made for the inheritance to be compatible with other vehicles, but the ground work is there.
As the design started to build itself, I felt that encapsulation became an increasingly important concept. Care was taken for my placement of methods, so I wouldn't break the code with the few protected variables that I had. Such protected variables included the dictionaries that stored data from the csv files.
The following diagram details the flow of actions for calculating the shortest route for a given list of airports, the main algorithm of the program. In this diagram, diamonds are decisions and boxes are actions.



Testing

To test this program, on top of manual debugging, entering in *print* statements to see what was going on. I created four unit tests. These tests can be found in the directory named **_unit_test.py*. The four tests were created to see if the code handled as expected.
aircraft_unit_test.py inspects whether the airports filled correctly from the *json* file. Initially, I wrote this test after I parsed the data direct from the csv file, so it was useful to see if my implementation worked as expected.
itinerary_unit_test.py checks if the routes and aircraft's loaded correctly from a predetermined test file, which is located in the main directory. It is called *testroutes.csv*. This test file can then be used as a template to create a multitude of routes to be passed through the program.
atlas_unit_test.py simply tests if the mathematics used to calculate the distance between two airports is operational.
currency_unit_test.py tests whether the algorithm to attach a currency object was successful. This test may also be view as a small system test, as it test to see if the interaction between different objects is as expected.
Ideally, a full black box test of this program would utilise a script to input all combination of airports to see if everything is working as expected.