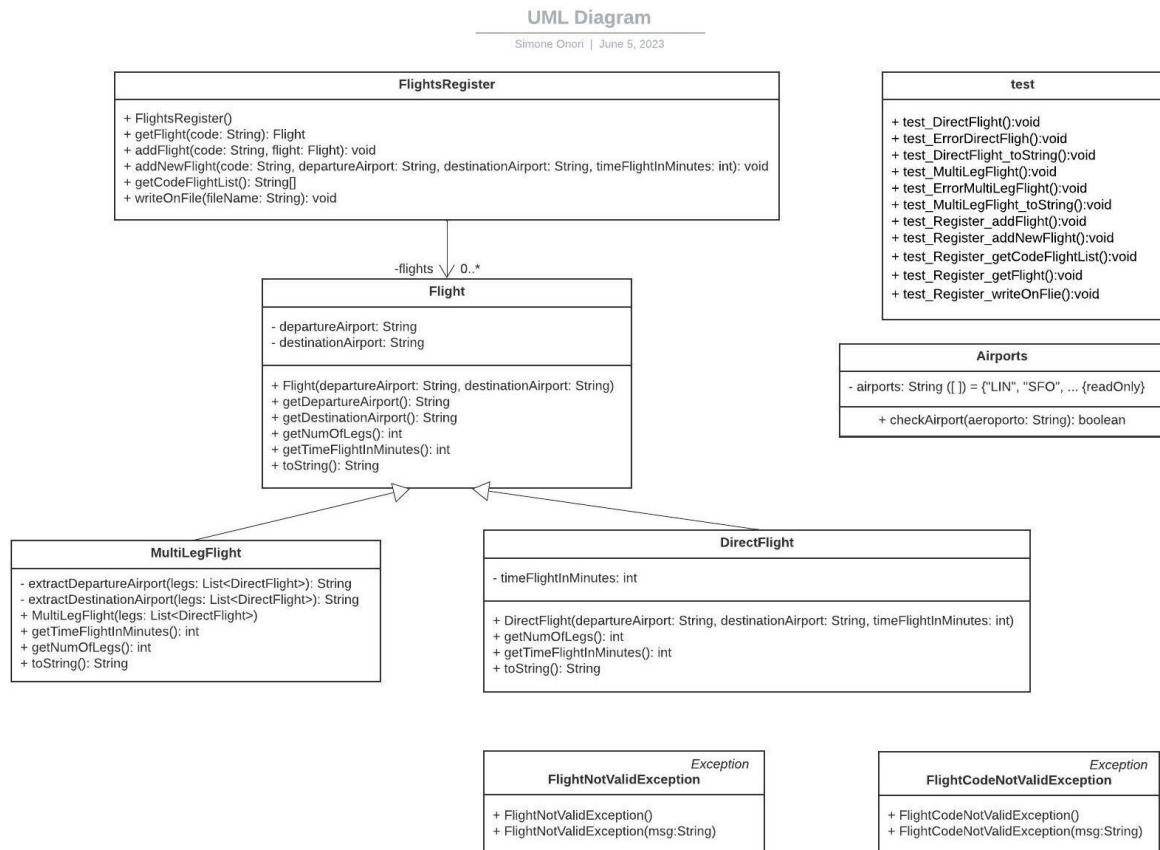


# Project Python Programming

Onori Simone and Vittorio Marco

We were thinking about an application that can handle civilian flights and that take into consideration the case in which the connection between the departing airport and the arrival airport is made with a single air route (direct flight) or it is expected and intermediate airport layover (multi leg flight).

The system implemented is represented in a more detailed way in the next UML diagram:



## Classes: Flight, DirectFlight, MultiLegFlight

The class *Flight* represent all the civil air flight managed by the system, that is direct flights (without intermediate airport layover) and multi leg flights (that expect one or more intermediate airport layover), organized by a generalization-specialization hierarchy in which the class *Flight* represents both types of flights in a hierarchical way.

We implemented 5 classes with necessary relationship in order to optimize the management of common characteristics to different types of flights and foster the polymorphic use of objects of the *Flight* type, taking into consideration that:

- Every flight is characterized by: a departure airport and a destination airport, the number of the air routes expected and the overall duration of the flight. Those four characteristics are immutable and accessible with get methods (see the diagram).
- A direct flight is instantiated with the constructor of the *DirectFlight* class, providing as parameters the airports, of departure and destination, and the flight duration in minutes. By definition, this type of flight always consists of only one leg. In order for the flight to be validly constructed, the duration of the flight must be a positive number and the departure and destination airports must be verified according to the *Airports.validAirport(String)* method.
- A multi-leg flight is instantiated with the constructor of the *MultiLegFlight* class, providing as a parameter a list (List) of *VoloDiretto* objects, which represents the set of single legs (each understood as a direct flight) to be flown. The duration of a *MultiLegFlight* is obtained as the sum of the durations

of all the scheduled sectors. For the flight to be validly constructed, there must be two or more legs and there must be a correspondence between the arrival and departure airports of the legs that follow one another in the input list.

- If the parameters passed to one of the constructors do not correspond to a valid flight as provided above for each type of flight, the constructor in question raises an exception of type *FlightNotValidException*.
- For each flight, the *toString* method returns a string describing the flight. The returned string include: departure airport, destination airport, the number of routes envisaged and the total duration of the flight.

### Classes: FlightRegister, Airport

It represent the register of the available flights. We used a dictionary to represent the association between the register and the flights: every flight is identified by a String type code that represent the access key to the dictionary.

The *addFlight(code, flight)* method adds a flight to the registry using the code passed as a parameter as the access key in the dictionary. The code must be a string exactly five characters long; otherwise the flight returns an exception of type *FlightCodeNotValidException*.

The *addNewFlight(code, departureAirport, destinationAirport, duration)* method is an “overload” of the above method for adding a new direct flight to the registry. The flight is created based on the passed parameters and it must satisfy the same constraint as above.

The *getFlight* method returns a flight given its code, or *None* if no flight exists with the given code.

The method *getCodeFlightList* returns an array of strings which contains all and only the codes of the flights present in the register.

The *writeOnFile(String fileName)* method writes all the flight log elements to the file *fileName*. Each item occupy a different row and the code is separated from the flight by a space. The method can throw exceptions of type *IOException*.

### Airport Class

The Airports class is a Python class designed to manage a list of airports and provide a method for checking the existence of a given airport code. This class can be used in various applications that involve airport-related operations, such as flight booking systems or travel management systems.

The Airports class consists of the following components:

- **Initializer Method (init):** This method is called when an instance of the class is created. It initializes the class by creating an instance variable called "airports" and assigns it a list of airport codes. The initial airport codes in the list are 'LIN', 'SFO', 'FCO', 'MXP', 'BGY', 'KRK', 'JFK', 'WAW', and 'MAD'. These codes represent a subset of airports used for demonstration purposes.
- **checkAirport Method:** This method allows users to check whether a given airport code exists in the list of airports. It takes an argument called "airport" which represents the airport code to be checked. The method iterates through the list of airports and compares each code with the given airport. If a match is found, the method returns True, indicating that the airport code exists in the list. If no match is found after checking all the airports, the method returns False, indicating that the airport code does not exist in the list.

### Test Class

Class with the tests to validate the implemented code.

### Exceptions

Classes used to manage all the exceptions generated by running the code.

## APP Class

The provided code represents a GUI application built using the Tkinter library in Python. The application is designed to manage flights and their details through a user interface. Here's a breakdown of the different components and functionalities of the code:

### Components:

- **Treeview Widget:** A table-like widget used to display flight information. It has columns for Flight Number, Departure Airport, Destination Airport, Legs, and Duration in Minutes.
- **Label Frames:** Two label frames are used to group related input fields and labels. The first label frame is named "Flight Details" and contains labels and entry fields for Flight Number, Departure Airport, Destination Airport, and Duration. The second label frame is named "Leg Airports" and is initially hidden. It appears when the user selects the "Multi-Leg Flight" option.
- **Labels and Entry Fields:** Various labels and entry fields are used to capture flight details such as Flight Number, Departure Airport, Destination Airport, and Duration.
- **Buttons:** Several buttons are present for different operations, such as adding a flight, clearing the entry fields, searching for a flight, retrieving a flight code list, and saving flight data to a file.
- **Checkbutton:** A check button labeled "Multi-Leg Flight" allows the user to indicate whether a flight has multiple legs. When selected, it dynamically generates additional entry fields for leg airports and their corresponding durations.

### Functionality:

- The `add_flight` function is called when the "Add Flight" button is pressed. It gathers flight details from the entry fields, creates a flight object (either a `DirectFlight` or `MultiLegFlight` depending on the selected mode), adds it to the `FlightsRegister`, and inserts the flight details into the treeview widget.
- The `clear_entries` function clears all the entry fields.
- The `copyOn_file` function writes the flight data to a file specified by the user in the "File Name" entry field and opens the file.
- The `searchFlight` function retrieves the flight details for a given flight code entered in the "Flight Code" entry field and displays the information in a message box.
- The `toggle_textboxes` function is triggered when the state of the "Multi-Leg Flight" check button changes. It shows or hides the leg-related entry fields based on the selection.

### Testing Approach

The Flight Management System project follows the Test-Driven Development (TDD) approach, ensuring code quality and comprehensive testing coverage. The key testing steps include:

- Writing tests before implementing the code for each class and functionality.
- Testing the creation of valid and invalid flight objects, ensuring proper validation checks.
- Verifying the expected behavior of flight retrieval, code listing, and file writing functionalities.
- Conducting rigorous testing to cover edge cases and potential error scenarios.

### Summary:

This code implements a graphical user interface (GUI) application for managing flights. It allows users to add flights, search for flights, view flight codes, and save flight data to a file. The application offers the option to handle multi-leg flights by dynamically generating additional entry fields for leg airports and durations. The Tkinter library is utilized to create the GUI components and manage their interactions.

**Bibliography:**

1. Introduction to Python for Computational Science and Engineering (A beginner's guide), Hans Fangohr, 2015
2. Computational Science and Engineering in Python, Hans Fangohr, 2016
3. Python GUI Programming with Tkinter, Alan D. Moore, 2018
4. UML @ Classroom, Martina Seidl, Marion Scholz, Christian Huemer, Gerti Kappel, 2015