College of Computing Technology
Faculty of Information Technology
Object Oriented Constructs

College Dublin
Computing • IT • Business

Application Development Project - Airplane

Miguelantonio Guerra - 2016324

Group B

Lecturer: Profesor Mark Morrissey

March 31, 2018

Contents

List of Figures

List of Tables

Chapter I.

The Problem

1.1. Problem

We were given the problem to design an airline schedule that will contain at least the following classes:

- Class AirPlane
  - Attributes:
    - make
    - model
    - capacity
    - pilot
  - Methods:
    - Constructor
    - getMake()
    - getModel()
    - getPilot()
    - capacity()
    - assignPilot()
    - toString()
- Class Flight
  - Attributes:
    - origin
    - destination
    - departure time
    - arrival time
    - date of flight
    - aircraft assigned
  - Methods:
    - Constructor

    - ∗ toString()
    - ∗ void schedule(String arrivalTime)
    - ∗ void schedule(String arrivalTime, String departureTime)

- Class CCTAir (main method)
  - Declare and initialize a number of flights
  - Add up to 5 flights
  - Use version 2 of Schedule method to set the time schedule for a flight
  - Use the version 1 of Schedule method to update the arrival time for a flight
  - Display all flights

This problem has to be solved using Object Oriented design applying the object oriented programming principles. We are also required to summit API documentation.

1.2. Solution to Problem

Now, before starting coding we set our rules for a classes, this means that we developed two interfaces, one for the AirPlane class and another for the Flight class. Other thing that we notice was that the pilot could be another object that will have a name and a license.

If we want to apply this problem to the real world an airline has different types of employees, such as, flight attendant, pilots, chefs, accountants, etc, so, we created a Super class called Staff from which the class Pilot extends its attributes (ID, name) and methods. The the extra attribute will be the license the pilot has.

After the interfaces were designed and the new two classes, Pilot and Staff, were drafted we decided to make a more interactive system of the airline. For this we decided to create a Menu to visualize the airplanes and the pilots and to manage the flights. We draw the menu with the different options we wanted on it and from this we design an interface for it. After doing this we realised that we needed to add a few more methods in our previous two interfaces.

After planing all these interfaces and classes we divided the work load as follows:

Classes	Staff	Pilot	AirPlane	Flight	Menu	CCTAir	DataGenerator
Adelo					X	X	
Asmer	X	X	X			X	
Miguelantonio				X		X	X

Table 1.1.: Division of the developing of the classes between the three of us.

Chapter II.

Individual Role within the Project

2.1. Overview of the Class Flight

After dividing the task in order to develop our system, as shown in the figure 1.1, It was up to me to get a working and functional Flight class that follows was it was agreed in its interface. Within this interface we find the method shows in the table 2.1.

	InterfaceFlight	
Return type	Method's name	Parameters
void	schedule	String arrivalTime
void	schedule	String arrivalTime, departureTime
String	getOrigin	None
String	getDestination	None
String	getDepartureTime	None
String	getArrivalTime	None
String	getDateOfFlight	None
AirPlane	getPlane	None

Table 2.1.: Flight interface and its defined methods.

The first two methods shown in the table 2.1 are the one given in the brief of the project and the rest are the ones that we decided to add in other to get the program we designed in our meetings. Now, this class has to have so me characteristics that describe it, because as for what we have in the interface are only behaviors. This attributes are listed as follows:

- A flight ID (int) to differentiate every flight we create to each other. This will function as a primary key if we make an analogy with databases.
- The origin (String) where the flight is leaving from.
- The destination (String) where the flight is going to.
- The departure time (String) of the flight.
- The arrival time (String) of the flight.
- The date of the flight (String).

- The plane (Object AirPlane) that is used for this flight.

After all this, we have everything that is necessary to create the class Flight.

2.2. Creating the Class Flight

Now, we must ask the question, How do we create an object of type Flight with the attributes mention in the previous section? We have to create a constructor as the one shown in the figure 2.1.

```
public Flight(String origin, String destination, String dateOfFlight, AirPlane plane) {
    flightId = id;
    id++;
    this.origin = origin;
    this.destination = destination;
    this.dateOfFlight = dateOfFlight;
    this.plane = plane;
    departureTime = "NOT AVAILABLE"; // Departure time is not assign when class is insta
    arrivalTime = "NOT AVAILABLE"; // Arrival time is not assign when class is instatiat
}
```

Figure 2.1.: Constructor for the class Flight.

As shown in the above figure this constructor only takes the origin string, destination string, date of the flight string and an airplane object. The ID is automatically created for each flight we are adding to our system and the fields departureTime and arrivalTime can be modified by the schedule method, but at the moment of creating a flight this two are unknown.

When a flight is added to the system we do not know when this flight is going to depart or when is going to arrive. For this we have the method schedule in two versions, the first one to update the arrival time and the second to set both arrival and departure times. This two methods are shown in the figure 2.2.

```
@Override
public void schedule(String arrivalTime) {
    this.arrivalTime = arrivalTime;
}

/** This method is used to set the arrival time and departure tim
@Override
public void schedule(String arrivalTime, String departureTime) {
    this.arrivalTime = arrivalTime;
    this.departureTime = departureTime;
}
```

Figure 2.2.: Schedule method version 1 and 2.

We can see that the version 1 takes the arrival time string as a parameter to update the arrival time and the version two takes two strings, arrival and departure time, to update both arrival and departure time.

Now, we were said to print each flight we a certain format (See briefing of the project) and for this we have to tell the print statement how to handle Obejcts of type Flight by overriding the toString method. The figure 2.3 shows how this method was coded. After this the command System.out.print() will know how to print a Flight Object in a format specified by us.

```
@Override
public String toString(){

    return "Fligh Information: \n\tDate: <<" + getDateOfFlight() + ">>"
            + "\n\tFrom: <<" +  getOrigin() + ">> to <<" + getDestination() + ">>"
            + "\n\tFlight Time: <<" + getDepartureTime() + ">> to <<"
            + getArrivalTime() + ">>\n" + getPlane();
}
```

Figure 2.3.: toString method to tell the print statement how to handle Objects of type Flight.

Finally, the other five methods that return a string were very straight forward. They are just going to return strings for the origin, destination, arrival time, departure time and date of the flight. And one only method is left, the getPlane method. Since a Flight receives an Object of type AirPlane and stores it as a attribute we might as well design a method that can return the whole object and for this we have the getPlane method. This last six methods were created in other to be able to access the attributes of a flight via methods instead of taking them directly from the attributes (these are declared as private so they can only be used within the class, in other words we use encapsulation on them). These six methods are widely used by the Menu class to be able to do a menu with sub menus and to access certain fields we wanted in there from the class Flight.

2.3. Creating the Data

Given that we were not working with databases we needed a way to generate the data to be use in our system. For this we had two options:

- Hard code the values into our program so this values can be use by the different classes in it.
- Using serialization to covert our objects into a stream of bytes and store this into a file.

We decided to apply the former creating a new class called Data to generate pilots and airplanes objects. In fact, this class has two methods, one that generates an ArrayList of Objects of type Pilot and the second that depends on the first one that generates an ArrayList of Objects of

type AirPlane. This values are then called by the main method (CCTAir) to generate another ArrayList of Objects of type Flight. The figure 2.4 shows how this class was created.

```
public class Data {

    /** This method creates and returns an arraylist of Objects Pilot ..
    public ArrayList<Pilot> createPilots() {...15 lines }

    /** This method creates and return an arraylist of Obejcts AirPlane
    public ArrayList<AirPlane> createAirplanes(ArrayList<Pilot> pilots)

}
```

Figure 2.4.: Data class to generate Obejcts Pilot and Objects AirPlane.

2.4. The Main Class - CCTAir

In the main class we made an combined effort to achieve what was asked in the briefing of the project. In this class first we generate the data for the pilots and the airplanes, then we initialized the flight arraylist and add 5 flights, and later on we used the schedule methods, version 1 and 2, to set the schedule of a flight and to update the arrival time of another flight. All this is shown in the figure 2.5.

```
Data data = new Data(); // Instatiating the class Data to be able to generate
ArrayList<Pilot> pilots = new ArrayList<Pilot>(); // Creating an ArrayList of
pilots = data.createPilots(); // Filling the arraylist with pilots
ArrayList<AirPlane> airplanes = new ArrayList<AirPlane>(); // Creating an Arra
airplanes = data.createAirplanes(pilots); // Filling the array with airplanes

ArrayList<Flight> flights = new ArrayList<Flight>(); // Creating an ArrayList

// Adding the 5 flights to our system.
flights.add(new Flight("Dublin", "London", "10/05/2018", airplanes.get(0)));
flights.add(new Flight("France", "Madrid", "23/06/2018", airplanes.get(3)));
flights.add(new Flight("Caracas", "Paris", "14/07/2018", airplanes.get(5)));
flights.add(new Flight("Brussels", "Berlin", "30/04/2018", airplanes.get(2)));
flights.add(new Flight("Dublin", "Moscow", "20/03/2018", airplanes.get(1)));

// Second version of the Schedule method to set the schedule for a flight
flights.get(4).schedule("15:00", "10:00");

// First version of the Schedule method to update the arrival time for a flight
flights.get(3).schedule("19:17");
```

Figure 2.5.: Generating pilots, airplanes, flights, and using the two version of the schedule method in the flight class.

Now, the idea of designing a menu was to show it to the user, but before that we have to print all the flights. For this we use that option that we have in our Menu class by instantiating

it and the calling this method, as shown in the figure 2.6. After the user user is done with reading the flights he can choose to go back and then we will get our Main Menu, just called after calling the display all flights method.

```
// Display all flights
// Here we are using our class Menu to perform this
// you can experience our system.
Menu menu = new Menu(pilots, airplanes, flights);
menu.deleteMenu();
menu.cabecera("All Flights");
menu.subDisplayAllFlights();
menu.mainMenu();
```

Figure 2.6.: Display all flights and Main Menu calling.

Chapter III.

Conclusion

Even though every one of us had its own tasks to develop this program it was not going to go anywhere if we did not set rules at the beginning for the designing of the classes. This way of programming gives us a common goal to seek without every one following its own path and getting a common mess at the end of it.

The design of interfaces helped us to follow certain rules and to have a common ground to stand on. Finally, we did this project in a cooperative manner to realize what was asked in the briefing of the project helping each other when a problem was presented in front of us.