Airline Flight Reservation Server (AFRS) - Release 1

Design Documentation
Prepared by Team 1:

- Stephen Cook <sjc5897@g.rit.edu>
- Niharika Reddy <nxr4929@rit.edu>
- Joshua Cotton <jdc5443@rit.edu>
- Moisés Lora <mal3941@rit.edu>

Table of Contents:

I. Summary

Our team was assigned with the task of creating a designing and implementing an Airline Flight Reservation Server (AFRS). This system allows the targeted users to provide flight information to travelers, as well as create, store and delete passenger reservations.

In order to create a well structured consistent design, our team decided to construct five subsystems: the User Interface Subsystem, the Airport Information Subsystem, the Itinerary Subsystem, the Flight Information and the Subsystem Reservation Subsystem.

The User Interface Subsystem represents our principal class where the user is able to access the system and is introduced to several possible options to interact with the system. To allow us to provide an efficient way to implement this functionality, we decided to utilize the Command Design Pattern, thus allowing us to dynamically store our commands in objects and for them to be handled by an Invoker Class.

The Airport Information Subsystem allows the client to query information about an airport by inputting a specific airport code, and proceeding to display its corresponding information: Airport name, Weather, Temperature and a possible delay time. For this scenario, we decided to implement the Observer Pattern. Our rationale was that the weather and temperatures are subject to change since there are many alternatives, therefore, the user interface would be the observer, and be notified, anytime that the weather changes, so that if an airport is queried for, the proper weather and temperatures are shown.

The Itinerary Subsystem allows a client to make a reservation for an itinerary contained in the most recent query for flight information. The reservation includes the passenger's name as well as their corresponding details such as the total cost of the itinerary, and flight information (flight number, origin and destination airports, departure and arrival times) for each flight in the itinerary. We decided that the Composite Pattern fits very well with this subsystem as it allows the recursive creation of reservations regardless if it's an individual flight, a itinerary consisting multiple flights or a more complex itinerary consisting of various different itineraries.

The Flight Query subsystem will take in a users request for flights and create all valid itineraries for that path. The subsystem must search through all paths from an origin and destination and validate

them by confirming that time aggregation of the arrival of the first flight plus a possible delay and a connection time still allows for the passenger to create a second flight. We decided to implement the Strategy Pattern which allowed us to utilize different sorting algorithms. Each sorting type changes the behavior of the system and since the sorting algorithm is chosen by the user, we have a strategy pattern on our hands.

Finally, our Reservation Subsystem was designed in order to allow the creation of reservations, the retrieval of existing reservations and the deletion of existing ones. Since it operates in three different modes, we decided to implement the Command Pattern here as well, which allows us to construct objects representing the three operation modes. These commands isolate itself from the UI subsystem and this makes it possible to switch out the UI interacting with reservations without changing Reservation at all.

In conclusion, all these subsystems work in conjunction to properly address all the requirements of the AFRS.

II. Domain Model

This section provides a domain model for the project. It should follow the guidelines discussed in class and the design project activity sheets. For it to be readable, you may need to turn this page into landscape mode.

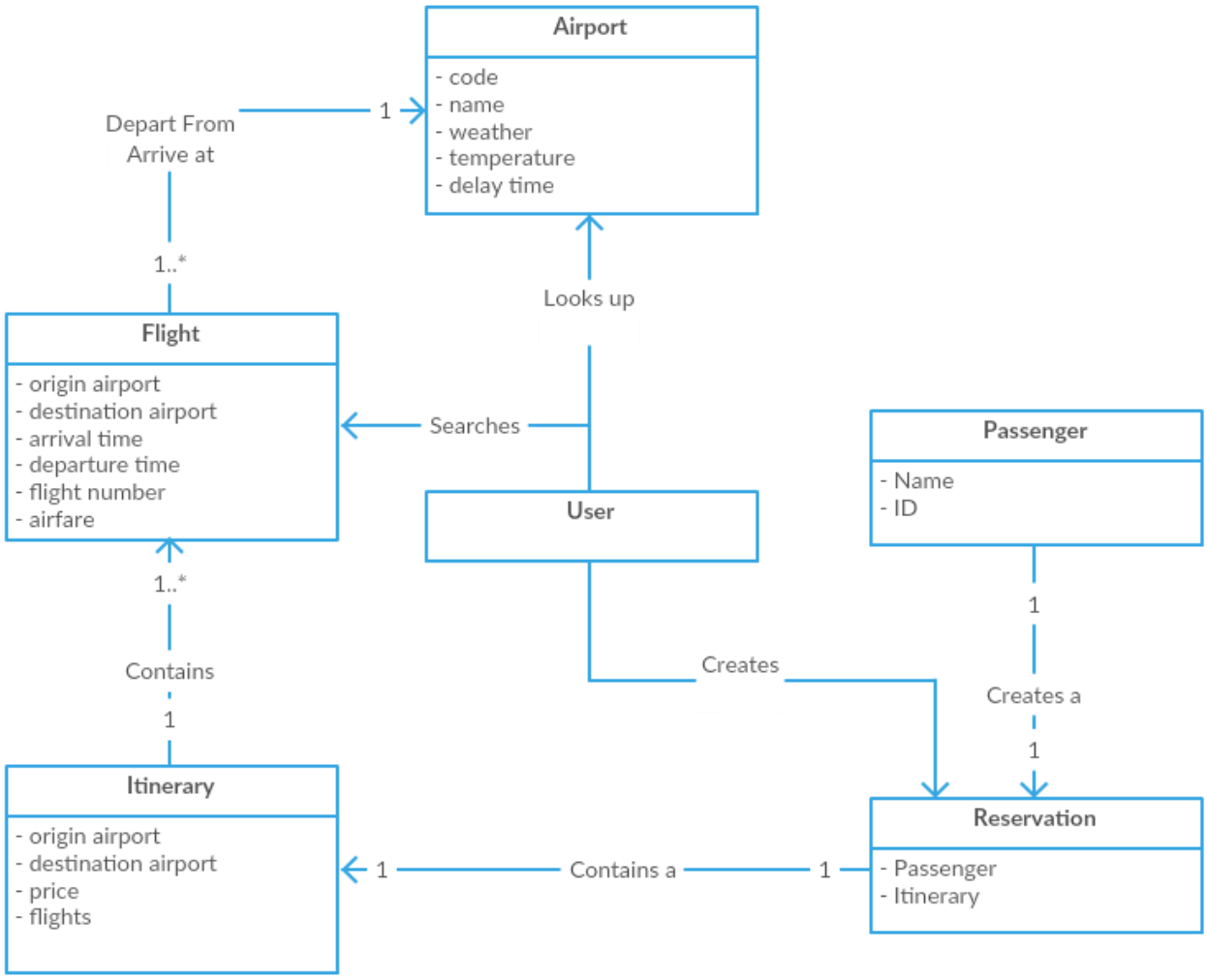

Figure 2-1: The Domain Model for the AFRS system

III. System Architecture

This section provides a model of the subsystem components that make up the overall software architecture for the project. Draw the subsystems as simple boxes with relationships between them. Provide a narrative that describes the responsibilities of each component and the interfaces that are provided between subsystems.

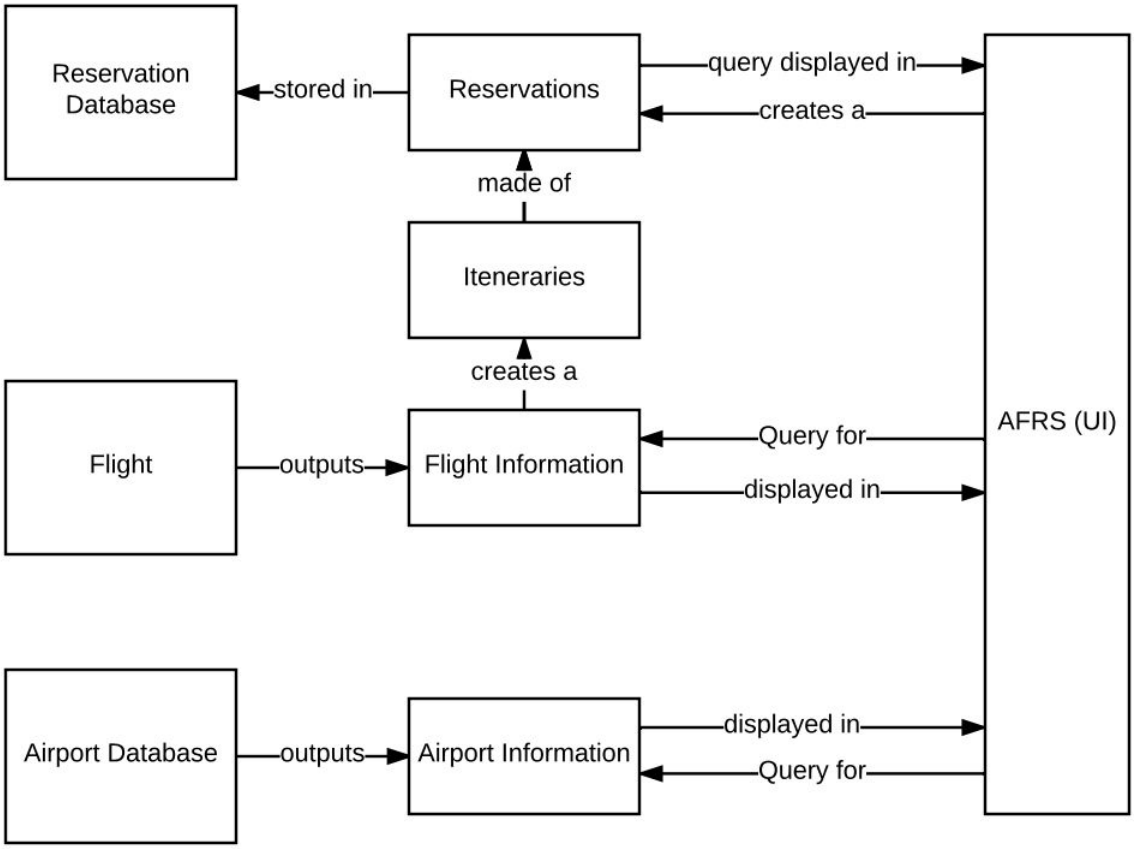

Figure 3-1: The System Architecture for the AFRS System

Design Rationale: Initially when creating the system architecture, we had a large database subsystem, where reservations, flight information, and airport information was stored in the same database. However, as we started to implement the system, we realized for this release it might be better to just have a small database or arraylist that just stores each subsystem separately for easier access within the class.

IV. Subsystems

A. User Interface Subsystem

Subsystem: UI
Depends on: all other subsystems

Description: The UI subsystem is responsible for text input parsing and passing the arguments to the text commands onto the individual command objects.

1. Read a line of input from the console
2. Split the split by on commas
3. If the request is not a partial request (the last part ends in a semicolon)
    a. Send the parameters to the command corresponding to the first entry on the line
    b. Print the strings returned by the command
4. Otherwise
    a. Report that a partial request is given
5. Repeat

Design rationale:

- *Command pattern*: the command pattern allows for commands to be added to a HashMap<String, Command> without having to update the UI implementation to be aware of the new command.
- *AFRS class*: the AFRS class that implements the UI was designed to be as simple as possible, and thus only contains the main method. Since there is only a HasMap of the command names and commands, and an input loop, it was deemed that a single-method class would be acceptable.

GoF Card:

Subsystem name: UI		**Pattern**: Command
Participants		
Class	**Role**	**Participant's contributions**
Command	Command interface	Defines the execute method for the commands
AFRS	Client	Creates the commands
AFRS	Invoker	Invokes the commands
FlightQueryCommand	Concrete command	Implements the logic to query for flights
CreateReservationCommand	Concrete command	Implements the logic to create a reservation
RetrieveReservationCommand	Concrete command	Implements the logic to retrieve an existing reservation
DeleteReservationCommand	Concrete command	Implements the logic to delete an existing reservation
AirportQueryCommand	Concrete command	Implements the logic to query for airport information
ExitCommand	Concrete command	Exits the AFRS when invoked
Reservation	Reciever	Handles the creation, deletion, and retrieval of reservations
AirportInfo	Receiver	Implements the actual logic to query for airport information
FlightQueryCommand	Receiver	Performs the query for flights
Deviations from the pattern	The AFRS class is both the client and the invoker, usually they're separate	
Requirements fulfilled by this pattern	Requirement 1	

Sequence diagram:

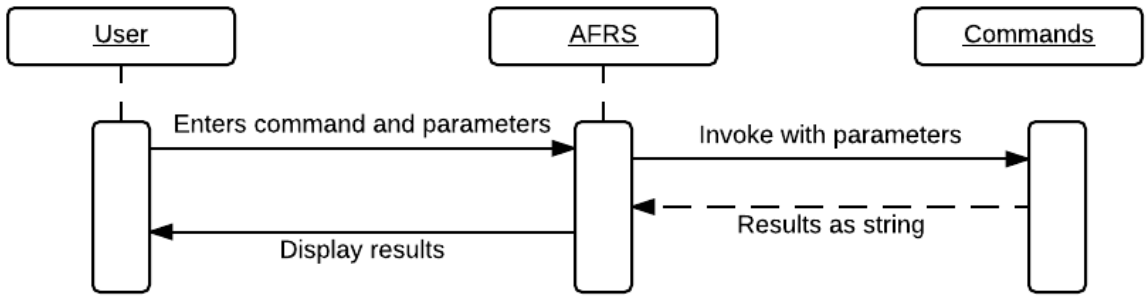

Figure 4A-01: Sequence Diagram for commands

UML Diagram:

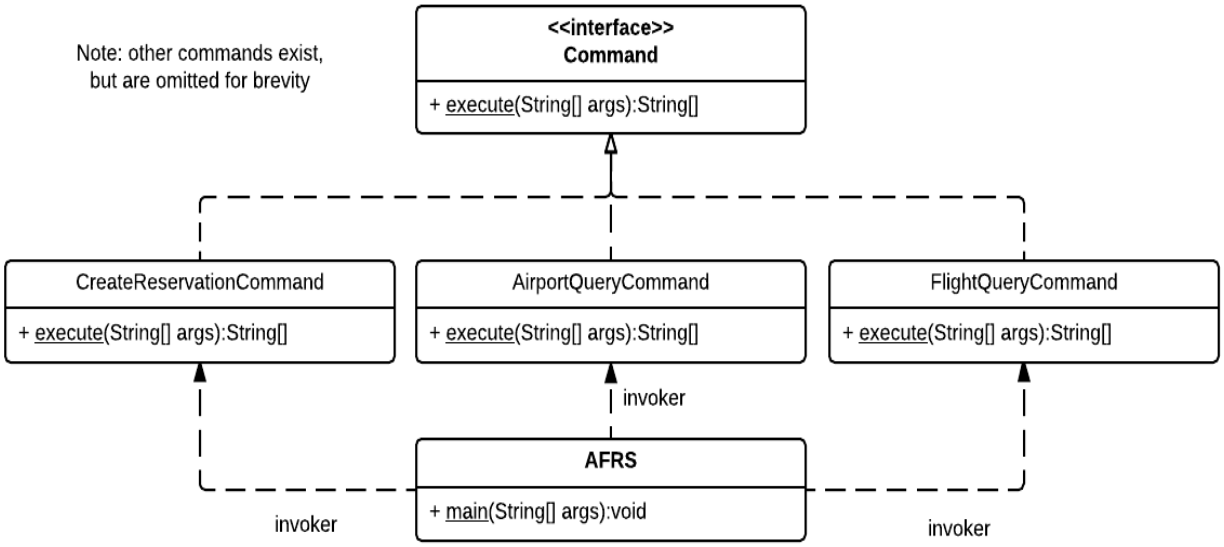

Figure 4A-02: Command and UI UML

B. Airport Query Subsystem

Subsystem: Airport Information

Requirements: The Airport Information subsystem shall allow a client to query for information about an airport (such as airport name, weather, temperature, and delay time) by inputting a specific airport code. If the client enters a null or invalid airport code, then the system will display an error message. After an airport code has been queried for, the system shall display (airport, [airportName], [airportWeather], [airportTemp], [airportCurrent Delay]).

Description: In order to query for airport information, the client shall enter in the text-string: “airport, “ followed by the three-letter code for the airport whose information is requested, and then end the request with a “ ; ” in the user interface.

1. Next, the system will read in the second input after the comma, indicating the airport code and pass it as an argument to AirportQueryCommand.
2. AirportQueryCommand will then call getAirport Information, to get the airport string that corresponds to the airport code.
3. getAirport Information will then read in a file of airport codes and corresponding airport names.
4. After parsing through the file, the system stores the airport name associated with the client-inputted airport code.
5. Next, the system will read in another file with the airport codes, and the corresponding weather and temperatures.
6. If there is only one set of weather and temperature, then the system will store that data into airportWeather and airportTemp.
7. If there are multiple datasets of weather and temperature associated with an airport code, then the system will iterate through the list and display a random dataset for each query result.
8. Next, the system will read in another data file to determine the delay time, which holds the airport code and the current delay time. The system will then store this number into airportCurrentDelay.
9. Finally, the information, will be displayed in the following format: (airport, [airportName], [airportWeather], [airportTemp], [airportCurrent Delay])

   airport, Orlando, sunny, 95, 15

10. If the airport code is unknown or null, then there will be an output of: error, unknown airport or error, null argument.

Design Rationale:

- *Observer Pattern*: It was difficult to find a pattern to specifically use for this subsystem, since, it is essentially reading in files and outputting the result of one line that corresponds in every line. However, we did eventually agree upon the observer pattern. Our rationale was that the weather and temperature is subject to change since there are many alternatives, therefore, the user interface would be the observer, and be notified, anytime that the weather changes, so that if an airport is queried for, the proper weather and temperature is shown.
- *AirportInfo class:* The team has a discussion over, what type of data structure should hold the airport information so that it can easily be accessible by other classes. Initially, we had decided to create an arraylist that would hold a airport code (String), airport name (String), a list of weather and temperatures (Arraylist), and delay times (int). However, later on in the week of implementation, it was discovered that a hashmap would be better suited for this type of implementation. However, there was not enough time to implement a working hashmap. This may replace the arraylist of airport information in release 2 of this project.
- We also discussed the law of demeter and separation of concerns, when thinking about having three different arraylists to represent each file, we eventually decided against it since it would be better to just have one array file and access a specific index of that one array list.

GOF Card:

<table>
<tr><td colspan="2">Subsystem Name: Airport Information</td><td>GoF pattern: Observer Pattern</td></tr>
<tr><td colspan="3">Participants</td></tr>
<tr><td>Class</td><td>Role in GoF pattern</td><td>Participant's contribution in the context of the application</td></tr>
<tr><td>AirportInfo</td><td>Subject</td><td>Once the AirportInfo has been queried for in the AFRS user interface class, then AirportInfo will send (“pushes”) the data payload of the new weather and temperature (once it has been updated) and notifies the AFRS (user interface)</td></tr>
<tr><td>N/A</td><td>Observer</td><td>We did not implement an observer interface since there is only one observer</td></tr>
<tr><td>AFRS</td><td>Concrete Observer</td><td>The client using the AFRS user interface, will make a query in this class about airport information and will pull from the subject (AirportInfo) the information that is needed (airport name, weather, temp, current delay time). This observer, will want to be notified of the weather and temperature, after it queries and it has been changed.</td></tr>
<tr><td colspan="3">Deviations from the standard pattern: There is a deviation in this pattern. Since there is only one observer (AFRS), we did not find it necessary to implement an abstract observer or interface for this pattern. The observer also gets notified only when it queries for airport information.</td></tr>
<tr><td colspan="3">Requirements being covered:</td></tr>
</table>

Requirement 1a, 1b, and 1c, Requirement 3i and 3ii

UML Diagram:

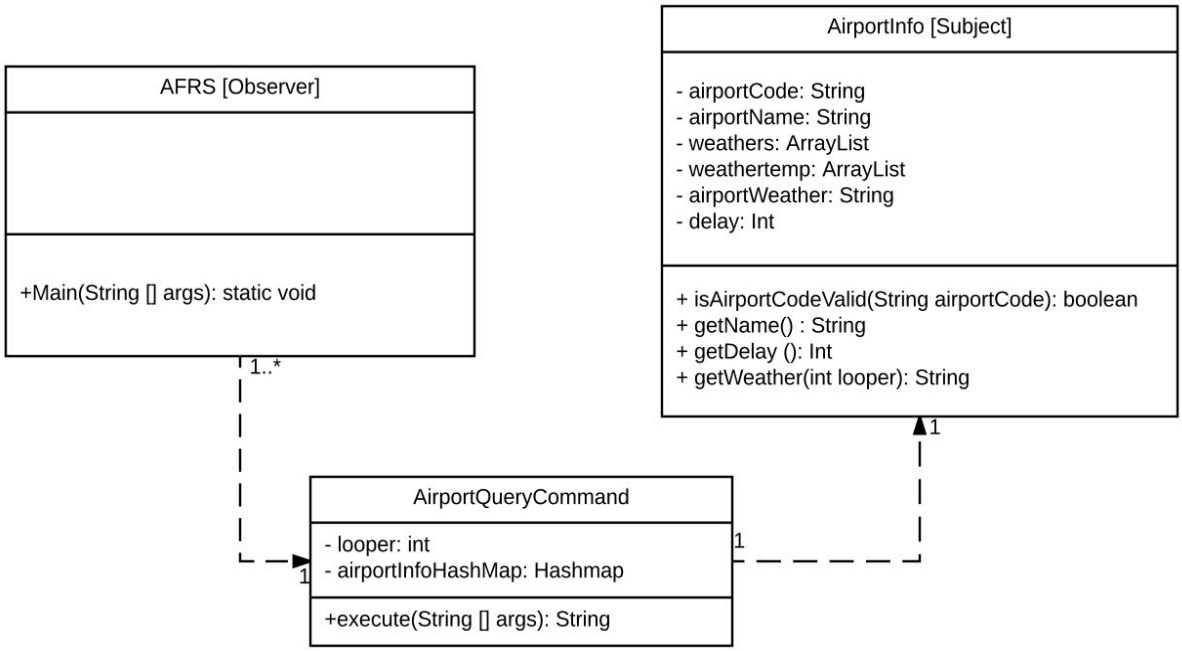

Figure 4.B-01: UML Diagram representing the Airport Information Subsystem

UML Diagram Note:

The AirportQueryCommand is not apart of the Observer pattern, it is just displayed to show that the AFRS does not make direct requests to AirportInfo since the User Interface (AFRS) implements the command pattern, but the Airport Information subsystem implements the observer pattern in a broader sense, since if AirportInfo changes, then the AFRS gets notified.

C. Itinerary Subsystem

Subsystem: Itinerary

Requirements: The itinerary subsystem shall allow a client to make a reservation for an itinerary contained in the most recent query for flight information. The reservation shall include the passenger's name along passenger's name along with the details of the reserved itinerary including the price for the itinerary, and flight information (flight number, origin and destination airports, departure and arrival times) for each flight in the itinerary.

Description: The subsystem allows the the system to create Itineraries based on the queried flights. Both the itineraries and flightInfo classes implement the itineraryinterface. This allows us to follow the composite pattern. The ItineraryInterface acts as the component; while, the Itinerary class acts as the composite. This means that the Itinerary must both implement and hold the Itinerary interface. Although this isn't used in this system, this does mean that an Itinerary can hold other Itineraries. Along with the normal getters, the Itinerary class has the addFlight method which adds children to the itineraries flight list. The other class important to the composite pattern is the FlightInfo class. The FlightInfo class holds all the information for one flight, like arrival and departure. This class acts as the leaf of the composite pattern and gets held in the Itinerary. We also have created a flights class that on startup is created and creates all the flight infos. This class then holds a list of all flights.

Design Rationale:

- *Composite pattern:* The composite pattern was chosen because it allowed us to create different length itineraries. Since an itinerary can hold up to three flights, the composite pattern allowed us to create a fluid class that could hold from one flight to three flights.
- *The Flight Class:* A small discussion was had over how to hold flight information in the system. The first was to create all flights on startup and hold them in system. Another idea was to create all itineraries at the start of system and just query the already created itineraries. The final was to create flights as they were queried. We went with the first one. The creating all itineraries at start up would make the startup very long and taxing and even with the potential benefits downstream the upfront cost would be too much. The third option would have saved upfront startup cost, but would slow down the rest of the system more. It was decided that we needed to create all flightInfo objects on start up and store them. This is what the flight class does. It creates them on start up and saves them in a state for call later in the life of the system.

GoF Card:

Subsystem: Itinerary		
Gof Pattern Name: Composite		
Class	**GoF Participant Name**	**Participant's activity within the pattern in the context of the application.**
ItineraryInterface	Component	The Component interface defines the common methods that the leaf and the composite classes are going to share.
FlightInfo	Leaf	The Flight class represents the most simple element of an itinerary and does not have any children.
Itinerary	Composite	The Itinerary class represents an element that has children whether it is flights or itineraries themselves.
Deviations from the standard pattern: The FlightInfo is also held in a Flight class. This isn't exactly part of the pattern, but it is used to store the flights		
Requirements being covered: 2B, 2D, 2E, 2F		

UML Diagram:

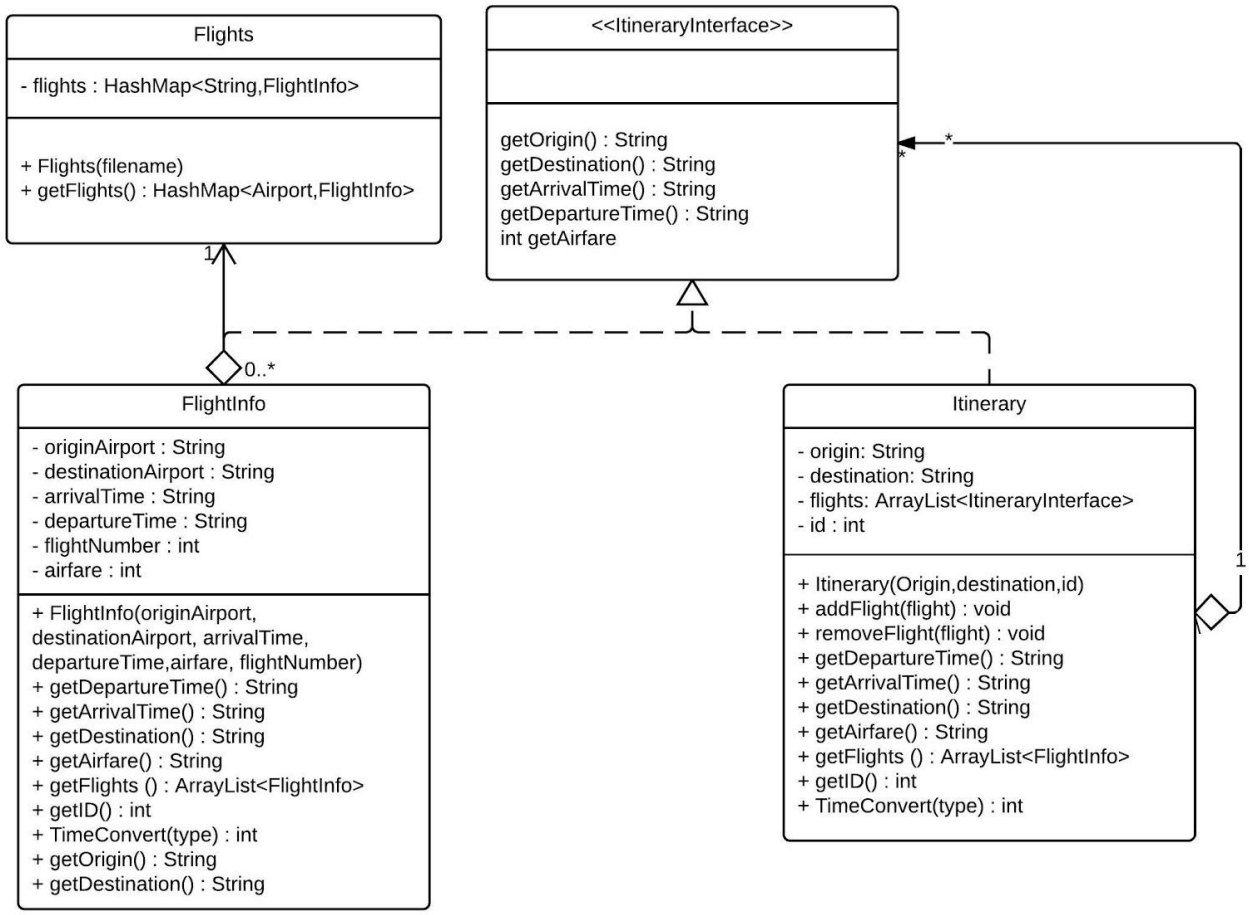

Figure 4.C-01: UML Diagram representing the Itinerary SubSystem

Sequence Diagram:

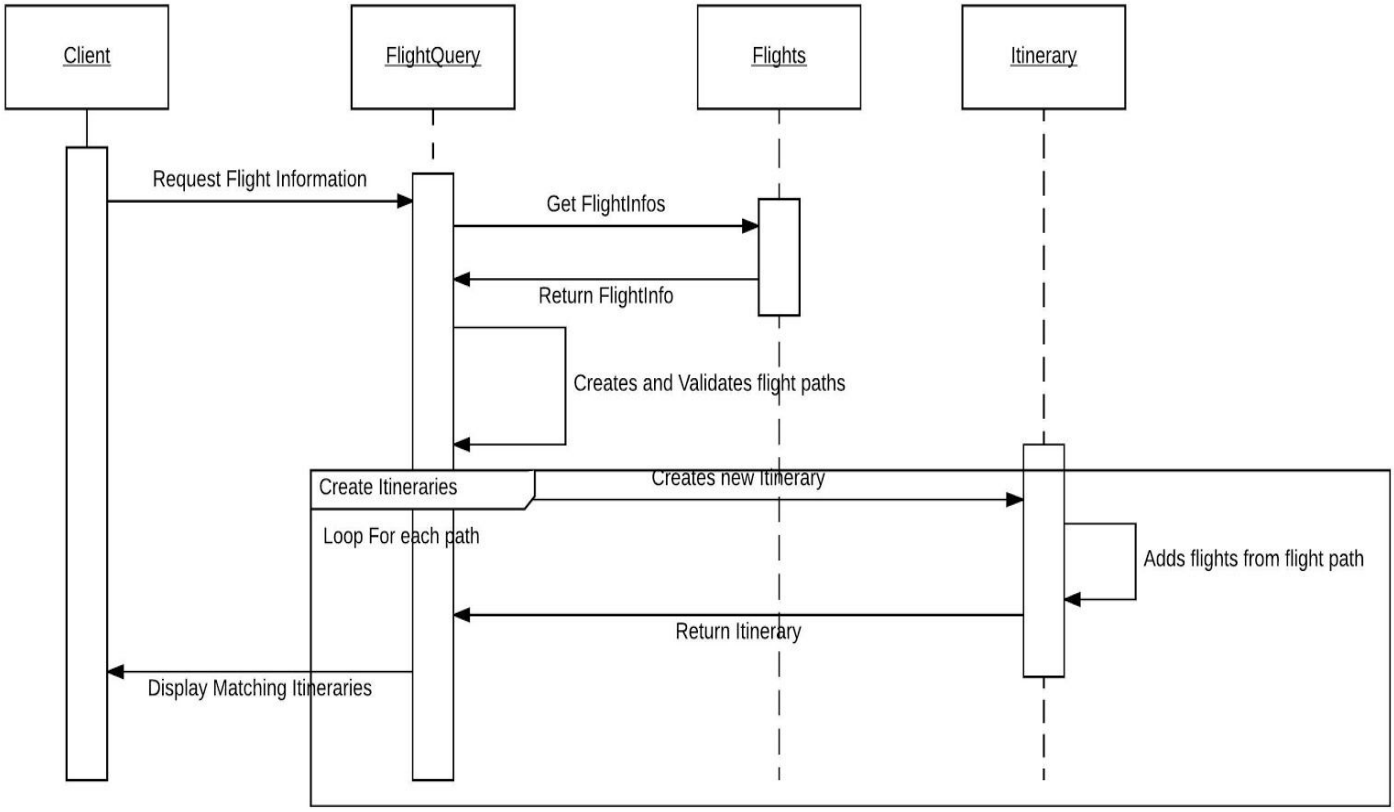

Figure 4.C-02: Sequence Diagram representing the Itinerary and Flight Query SubSystem

Sequence Description:

When the client request flight information it creates a new flight query object. This flight query object then gets all the flight information from the Flights class and creates and validates all the flight paths. Then with each correct flight path it creates a new Itinerary which adds all the flights from the found flight path. Finally the system then returns the valid Itineraries.

D. Flight Query Subsystem

Subsystem: Flight Query

Requirements: The Flight Query subsystem will take in a users request for flights and create all valid itineraries for that path. To do this the flight query system must search through all paths from an origin and destination and validate them. This validation is that the arrival of the first flight + delay + connection time still allows for the passenger to make the second flight. This subsystem also uses the strategy pattern to sort the queried flights by airfare, departure time, or arrival time.

Description: The flight query subsystem's hardest job is to find the valid flight paths for the inputted airports. This is done by getting the hash map of flights and starting at the origin. All flights from the origin are added to a list. If they end in the destination the are added to the success list, if not they are added to the path list. We keep going until the max number of connections are reached and all flights in the success list get validated for time in the isValid method. Once the flights are validated, each flight path gets made into an itinerary and returned to the UI. The returned filght query is also sorted based on user preference and using an insertion sort.

Design Rationale:

- *Strategy Pattern:* The different sorting algorithms are perfect for the strategy pattern. Each sorting type changes the behavior of the system and since the sorting algorithm is chosen by the user, we have a strategy pattern on our hands.
- *Hashmap over ArrayList:* Originally I used an arraylist to store all the flights. The system had performance problems when finding all possible paths out of the airports. I originally thought this was do to having to iterate over an arraylist over and over again, so I switched to a hashmap where the keys were the airport codes and the values were the flights. This helped performance slightly, but I still saw some problems with performance.
- *isValid:* At first I wanted to validate each path as I created it, but this proved too difficult in the time I had so I ended up creating an isValid method. This method checks the whole path to see if the times work out and to see if the flight end in the origin. Since this is checking so many flight paths this was where most of the optimization needs to happen in release 2.
- *SuccessPath*: In an attempt to help optimize the isValid and flightQuery, this was added. While creating a path if the system notices that it ends in the destination it adds it to this path and ends the path there. Then we return this list instead of the all possible path list. This greatly cut down

on the number of flights that isValid needs to test. When debugging before the change we got around 544,000 flight paths, after we only needed to check around 25,000 flight paths. This also reduced the time by almost 80% on some more taxing paths (ie SFO,LAX,2) .

GOF Card:

Name: Flight Sorting		**GoF pattern: Strategy Pattern**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
FlightQuery	Context	This class holds the itineraries of queried flights. This class also uses the Sorting Strategy to sort it based on the user defined sorting method.
Sorting	Strategy	This is the interface that the concrete strategies use. Here we can switch between sorting algorithms to fit the user's desired sorting method.
DepartureTime	ConcreteStrategy A	This holds the sorting algorithm for sorting by Departure Time.
ArivalTime	ConcreteStrategy B	This holds the sorting algorithm for sorting by Arrival Time.
Airfare	ConcreteStrategy |C	This holds the sorting algorithm for sorting by Airfare.
Deviations from the standard pattern:		
Requirements being covered: 2B, 2C, 2D, 2E, 2F		

UML Diagram:

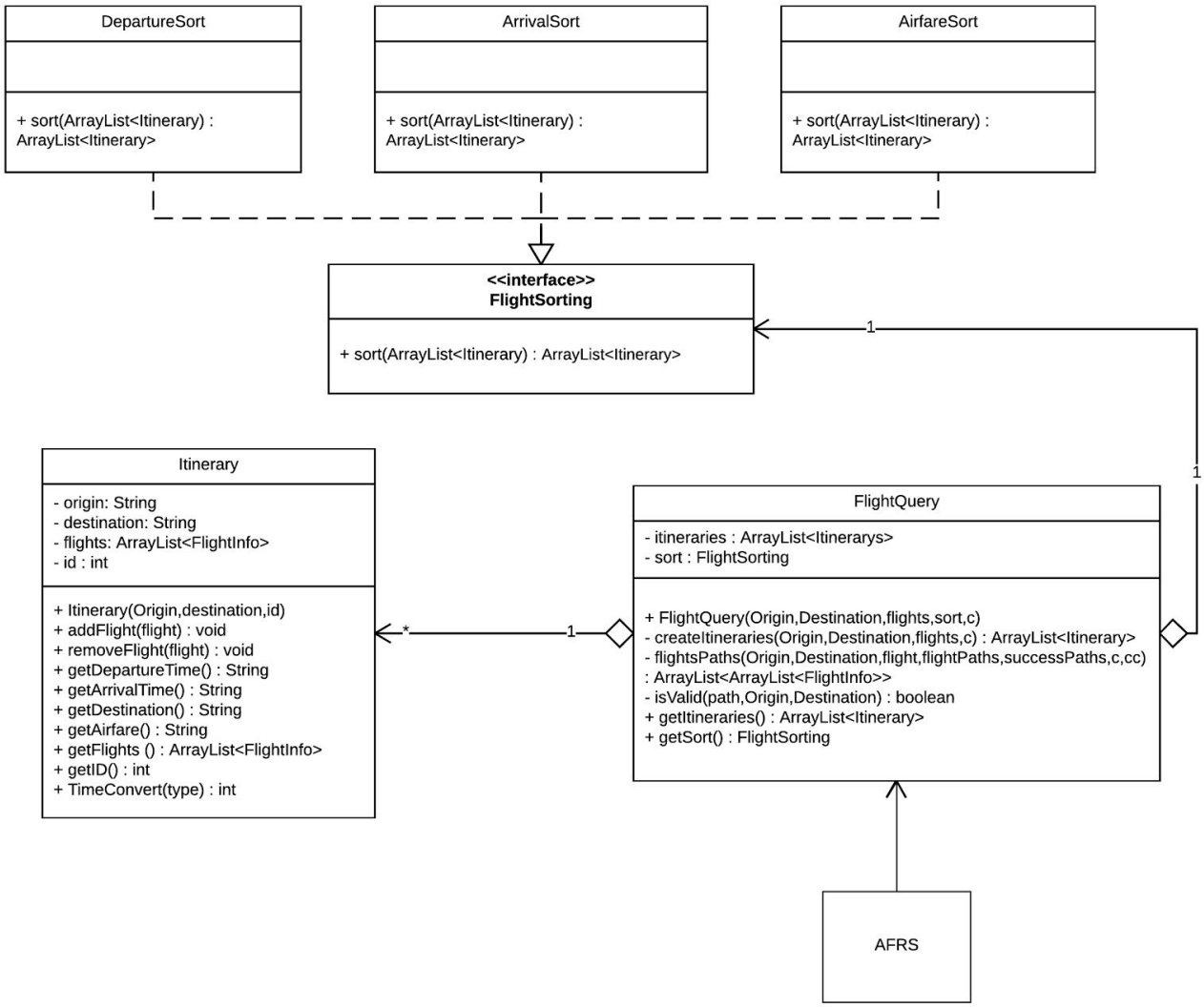

Figure 4.D-01: UML Diagram representing the Flight Query Subsystem

Sequence Diagram:

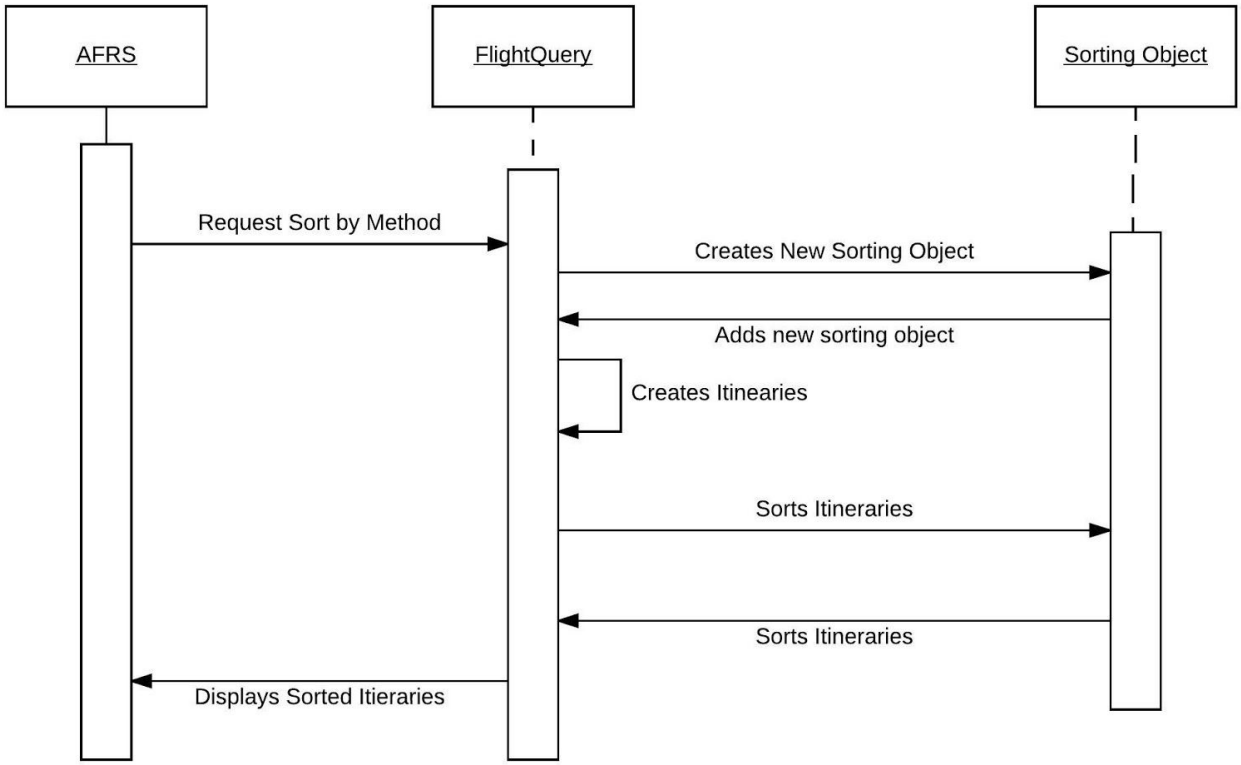

Figure 4.D-02 : Sequence Diagram for sorting Flights

Sequence Description:

The AFRS System request the flight information with the requested sorting method. In the example Sorting Object is any of the sorting objects, ArrivalSort, DepartureSort, or AirfareSort. The FlightQuery gets created and creates a new sorting object and stores it as a state. The flightQuery now creates all the Itineraries and sends them to the sorting object to be sorted. Then the sorting object returns them and the flight query returns the now sorted Itineraries. Finally the AFRS system displays the sorted Itineraries.

E. Reservation Subsystem

Subsystem: Reservations (**Depends on**: Flight Information subsystem)

Requirements: Requirement 4

Description: The reservation subsystem shall support three operations:

Creating a reservation:

Preconditions:

1. A flight information request that returned flights (and therefore created itineraries) has already been executed.

Inputs:

1. The ID of an itinerary created by the last flight information request.
2. The name of the passenger creating the reservation.

Steps:

1. Retrieve the itinerary corresponding to the passed itinerary ID. [1]
2. Create a Reservation instance passing the itinerary and passenger name to the constructor.
3. Compare this new Reservation instance to the already stored Reservation instances. [2]
4. If the new Reservation does not have the same name and origin/destination pair as an existing reservation, store it
    a. Otherwise fail

Postconditions:

1. The reservation is stored

Implementation notes:

1. Or should we just give the ID to the reservation?
2. A method to compare the passenger name and origin/destination pair of two reservations would be useful

Retrieving existing reservations:

Preconditions: None

Inputs:

1. Passenger name
2. [Optional] Origin airport
3. [Optional] Destination airport

Steps:

1. Retrieve the list of reservations from the reservation store
2. Filter the reservations by passenger name
3. If an origin airport was passed, filter the reservations by origin airport
4. If a destination airport was passed, filter the reservations by destination airport

5. Return the filtered reservations

Postconditions: None

Implementation notes:

1. A Java 8 Stream would be the easiest way to do filtering

Deleting an existing reservation:

Inputs:

1. The passenger name
2. The origin airport
3. The destination airport

Preconditions: none

Steps:

1. Retrieve the list of reservations
2. Filter the reservations based on the passed passenger name, origin airport, and destination airport
3. If a single reservation was returned delete that reservation from the system and report success
    a. Otherwise if no reservation was returned report failure

Postconditions:

1. The reservation is deleted from the system

Design Rationale:

- *Command pattern*: the reservation subsystem uses commands to isolate itself from the UI subsystem. This makes it possible to switch out the UI interacting with reservations without changing Reservation at all.
- *Reservation class*: the Reservation class' logic for creating, retrieving, and deleting reservations is in static methods. This allows these methods to be used without having an instance of a class, and furthermore they don't require an instance to begin with. Instances of the Reservation class represent an actual reservation in the system.

GoF Card:

Subsystem name: Reservations		**Pattern**: Command
Participants		
Class	**Role**	**Participant's contribution**
Command	Command interface	Defines the interface that commands will conform to
CreateReservationCommand	Concrete command	Implements the logic for creating reservations
RetrieveReservationCommand	Concrete command	Implements the logic for retrieving reservations
DeleteReservationCommand	Concrete command	Implements the logic for deleting reservations
AFRS	Client	Creates the command and sets its receiver
AFRS	Invoker	Invokes the command to carry out its logic
Reservation	Receiver	Contains the actual logic called upon by the commands
Deviations from the pattern: AFRS is both the client and invoker, usually they're separate classes		
Requirements fulfilled by this pattern: All of requirement 4		

UML diagram:

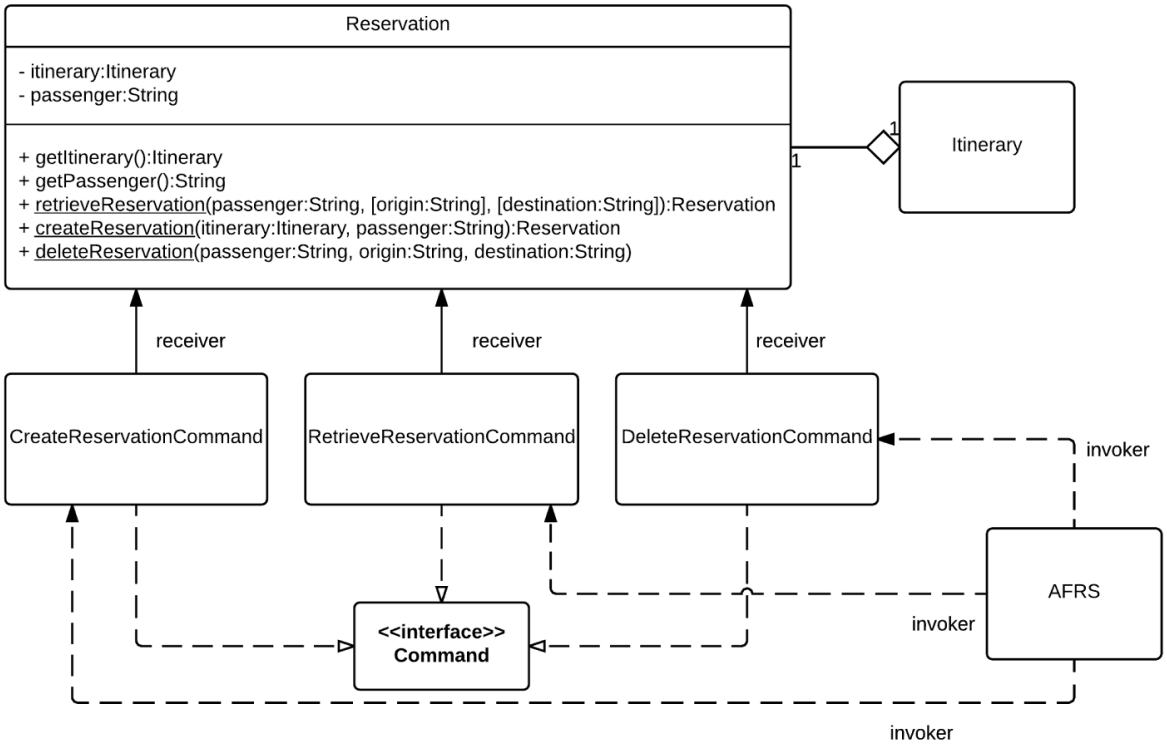

Figure 4.E-01: UML Diagram representing the Reservation Subsystem

Sequence diagrams

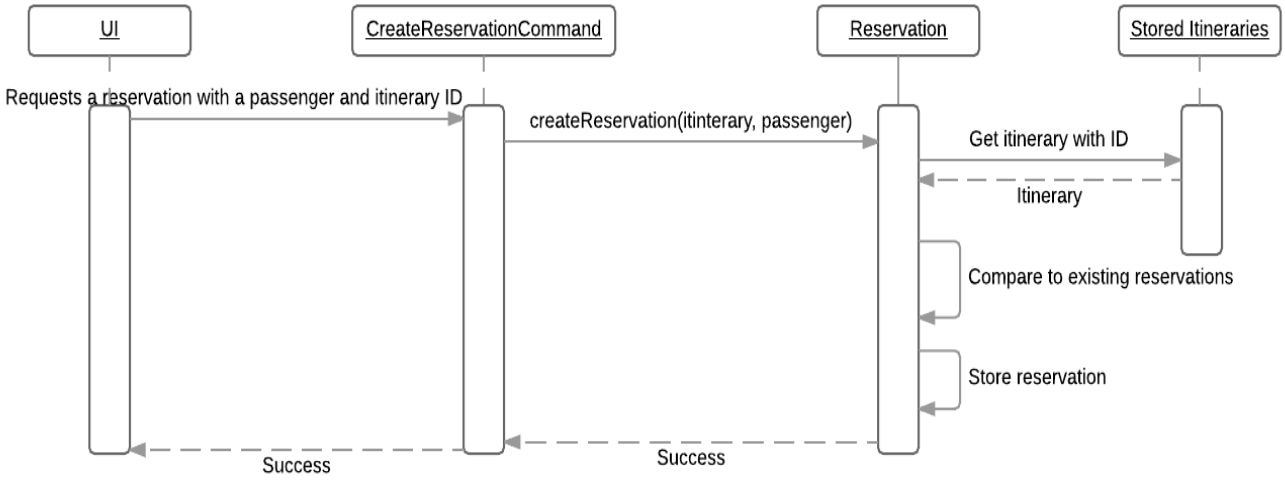

Figure 4.E-02: Sequence Diagram representing creating a reservation

Figure 4.E-03: Sequence Diagram representing retrieving an existing reservation

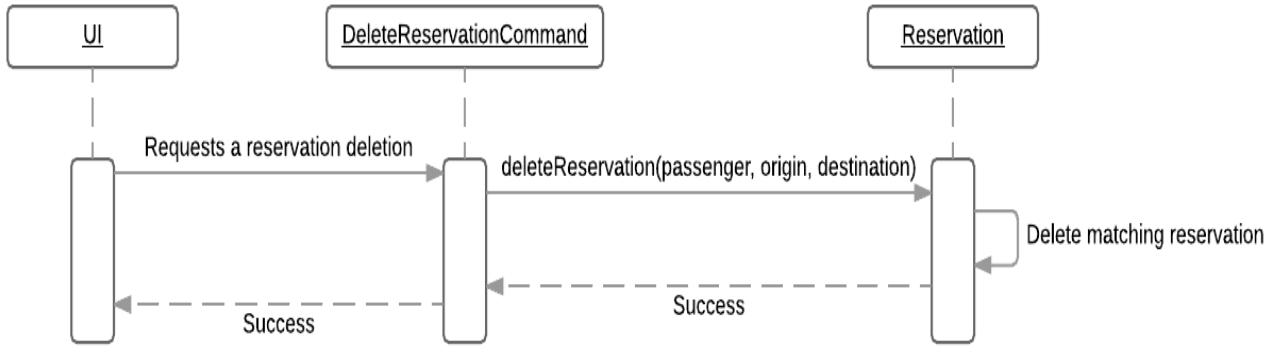

Figure 4.E-04: Sequence Diagram representing deleting a reservation

Status of the Implementation

Provide a complete description of the status of your implementation. This should specify all known defects in the system, and indicate requirements that your implementation does not cover.

User Interface Subsystem:

- Complete, with additional exit command

Airport Information Subsystem:

- The arraylist is not currently accessible outside of the class, currently returns string with the pertinent airport information.
- Exception of invalid airport code is handled by an isValid() method instead of being handled by an IllegalArgumentException Handler.
- All user requirements were met.

Flight Query Subsystem:

- Runs a little Slow for some larger airports (LAX) with larger connections (2)

Reservation Subsystem:

- Reservations are not persisted

Appendix

Class: AFRS	
Responsibilities: Implements the program entry point and text user interface	
Collaborators:	
Uses: AirportQueryCommand, CreateReservationCommand, Command, DeleteReservationCommand, RetrieveReservatoinCommand, FlightQueryCommand	**Used by:**
Author: Joshua Cotton	

Class: AirfareSort	
Responsibilities: This class holds the algorithm for sorting by airfare. It is an insertion sort and acts as a concStrategy in the Command Pattern	
Collaborators:	
Uses: FlightSorting	**Used by:** FlightQuery FlightQueryCommand
Author: Stephen Cook	

Class: AirportInfo	
Responsibilities: This class is used to read in three files and store the airport name, weather temperature, and temperature corresponding to each airport code. This class can be accessed to retrieve the information stored about an airport. It can also be called to display information about an airport.	
Collaborators	
Uses:	**Used by:** AirportQueryCommand
Author: Niharika Reddy	

Class: AirportQueryCommand

Responsibilities: This class is used to call the AirportInfo class, and return a string of the airport name, weather, temperature, and delay time of a specific airport code. If the user does not enter in anything, it will raise a null argument exception, if the client does not enter a valid airport code, it will raise an illegal argument exception.	
Collaborators	
Uses: AirportInfo	**Used by:** Command
Author: Niharika Reddy	

Class: ArrivalSort	
Responsibilities: This class holds the algorithm for sorting by arrivalTime. It is an insertion sort and acts as a concStrategy in the Command Pattern	
Collaborators:	
Uses: FlightSorting	**Used by:** FlightQuery FlightQueryCommand
Author: Stephen Cook	

Class: CreateReservationCommand	
Responsibilities: Implements the logic for creating a reservation.	
Collaborators:	
Uses: Command, Reservation	**Used by**: AFRS
Author: Joshua Cotton	

Class: Command	
Responsibilities: Represents the interface that commands implement	
Collaborators:	
Uses:	**Used by:** AirportQueryCommand, CreateReservationCommand, DeleteReservationCommand, RetrieveReservatoinCommand, FlightQueryCommand
Author: Joshua Cotton	

Class: DeleteReservationCommand	
Responsibilities: Implements the logic for deleting reservations	
Collaborators:	
Uses: Command, Reservation	**Used by**: AFRS
Author: Joshua Cotton	

Class: DepartureSort	
Responsibilities: ... This class holds the algorithm for sorting by Departure. It is an insertion sort and acts as a concStrategy in the Command Pattern	
Collaborators:	
Uses: FlightSorting	**Used by:** FlightQuery FlightQueryCommand
Author: Stephen Cook	

Class: ExitCommand	
Responsibilities: Exits the AFRS UI	
Collaborators:	
Uses: Command	**Used by**: AFRS
Author: Joshua Cotton	

Class: FlightInfo	
Responsibilities: ... This class is used to store the information of a single flight. It contains the origin, destination, airfine, departure, arrival, and flight number of each flight	
Collaborators:	
Uses:	**Used by:** FlightQuery Itinerary
Author: Stephen Cook	

Class: FlightQuery	
Responsibilities: ... This class queries all flight information and creates new itineraries based on users inputs. It also validates flight paths to insure that all flights along the path can be made	
Collaborators:	
Uses: FlightSorting FlightQuery	**Used by:** FlightQueryCommand
Author: Stephen Cook	

Class: FlightQueryCommand	
Responsibilities: ... This is the concCommand for querying flights that the command pattern uses. It will set up the flights on startup and parse the user input to allow for correct operation. It also parses the result for proper output	
Collaborators:	
Uses: Command, FlightSort,	**Used by:**
Author: Stephen Cook	

Class: FlightSorting	
Responsibilities: ... This is the interface for the sorting flight information Iteniraries. This class acts as the strategy in the strategy pattern	
Collaborators:	
Uses:	**Used by:** Arrival Sort, Departure Sort, Airfare Sort, FlightQuery
Author: Stephen Cook	

Class: Flights	
Responsibilities: ... This classes only responsibility is to create and store all the individual flightInfo objects	
Collaborators:	
Uses:	**Used by:** FlightInfo
Author: Stephen Cook	

Class: Itinerary	
Responsibilities: ... This class is used to represent the information of a single Itinerary. It holds the origin destination and list of flights. It acts as the Composite in the composite pattern	
Collaborators:	
Uses: ItineraryInterface	**Used by:** FlightQuery
Author: Stephen Cook	

Class: ItineraryInterface

Responsibilities: ... This is the interface that both the Itinerary and the FlightInfo implement to create a composite pattern.	
Collaborators:	
Uses:	**Used by:** Itinerary FlightInfo
Author: Stephen Cook	

Class: Reservation	
Responsibilities: Stores reservations and holds static methods for commands to use	
Collaborators:	
Uses:	**Used by:** CreateReservationCommand, DeleteReservationCommand, RetrieveReservationCommand
Author: Joshua Cotton	

Class: RetrieveReservationCommand	
Responsibilities: Implements the logic for retrieving a reservation based on a set of criteria	
Collaborators:	
Uses: Command, Reservation	**Used by**: AFRS
Author: Joshua Cotton	