Airline Flight Reservation Server (AFRS) - Release 2

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

The task assigned was the design and implementation of an Flight Reservation Server (AFRS). This system allows the targeted users to: provide flight information to travelers, and create, store and delete passenger reservations.

In order to create a well structured consistent design, it was decided that five subsystems would be constructed; the User Interface Subsystem, the Airport Information Subsystem, the Itinerary Subsystem, the Flight Information and the Subsystem Reservation Subsystem.

The User Interface Subsystem allows the user to interact with the server by the use of commands. In this release, the session now acts as the invoker of the commands due to the necessity of multiple connection ids. In addition, undo and redo functionality has been added to the commands, connect and disconnect commands have also been implemented and there is a new graphical user interface for the user to utilize.

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. In this release, the subsystem makes use of the strategy pattern by allowing the client to query either locally or through the FAA web service, which is handled by a remote proxy which pulls and parses the data.

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. This release saw no change to the Itinerary Subsystem.

The Flight Query subsystem allows for the client to view all the possible itineraries for certain destinations. The subsystem takes in a client's request for flights and creates all valid itineraries for that path. This release saw no change to the Flight Query subsystem

The Reservation Subsystem was designed in order to allow the creation of reservations, the retrieval of existing reservations and the deletion of existing ones. In this release, the persistence of reservations has been successfully implemented, and are able to be loaded upon the application's startup. In addition, the undo/redo functionality is now present for the user to utilize.

Finally, the Session Subsystem allows for and keeps track of the different connections that the AFRS must handle. This subsystem allows for flight queries and undo/redo stacks to be unique to a connection. It was decided for the state pattern to be used by the subsystem, as it allows to represent the two states of a session, connected and disconnected.

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

II. Domain Model

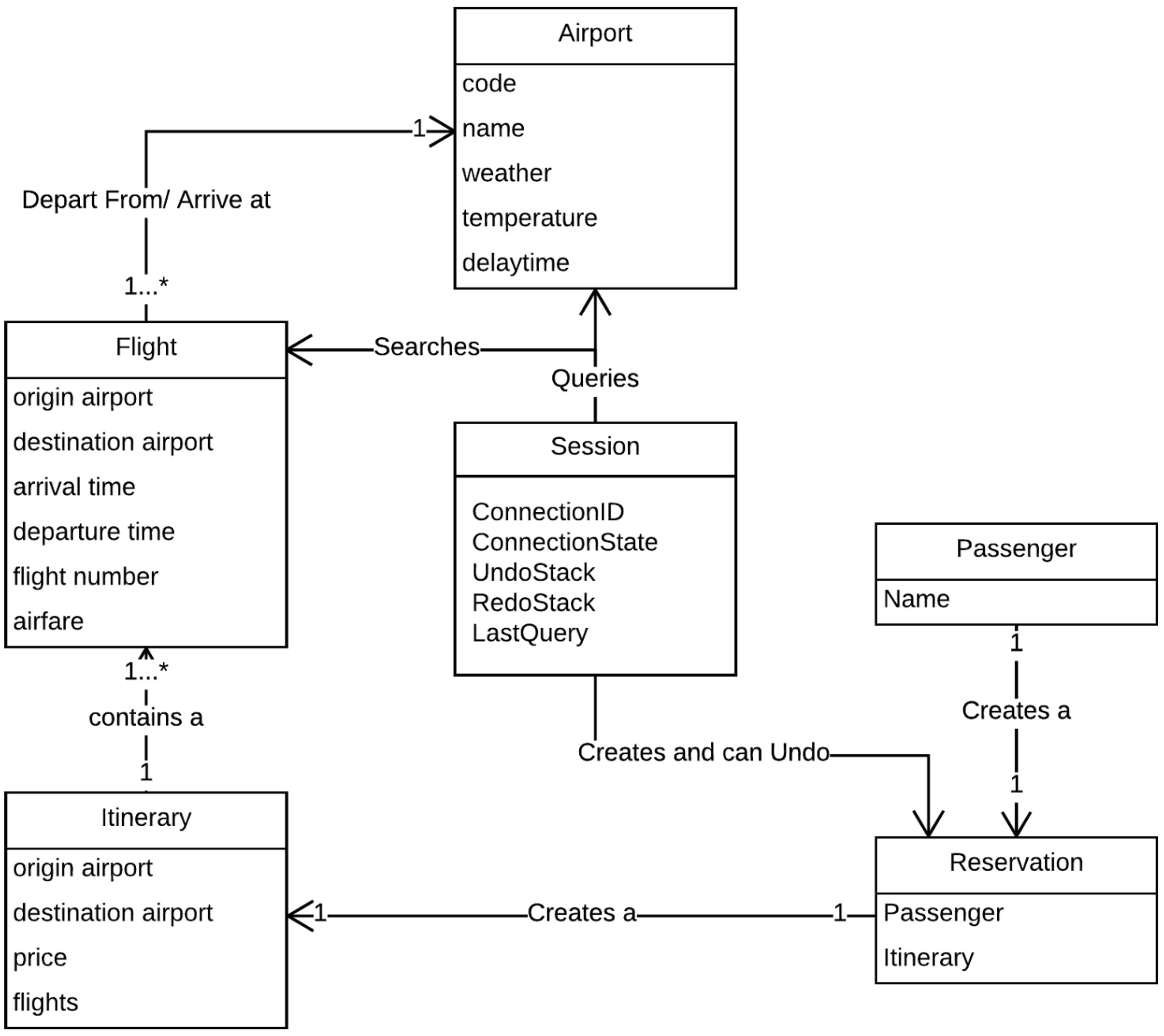


Figure 2-1: The Domain Model for the AFRS system

III. System Architecture

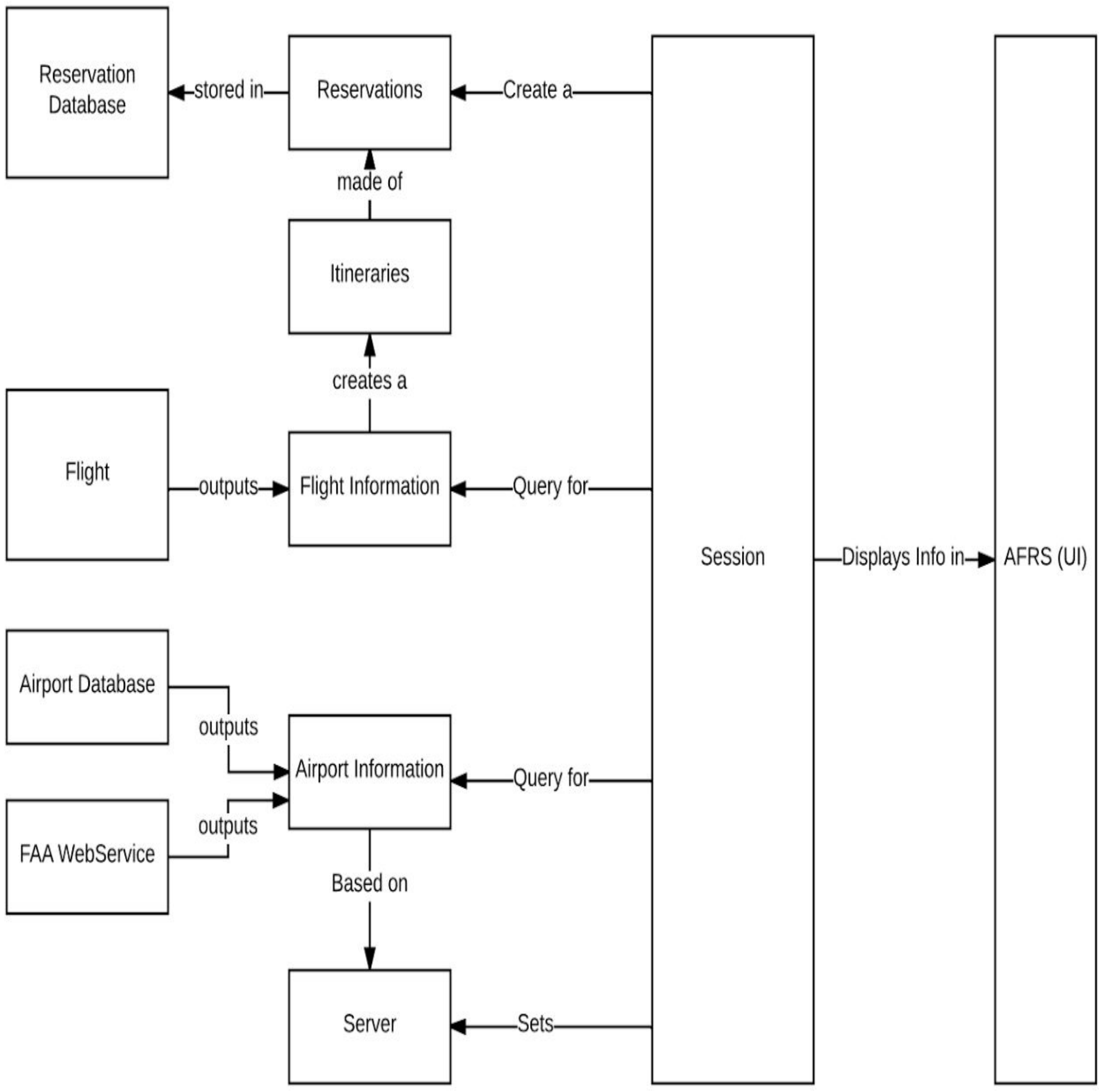


Figure 3-1: The System Architecture for the AFRS System

Design Rationale: Initially, when designing the system architecture, there was a large database where reservations, flight information, and airport information was stored in the same database. However, as the system design was implemented, separating the databases into smaller hashmaps that corresponded with the subsystem was chosen over implementing separate database classes. By moving to create hashmaps containing airport information, flight information, and reservations, cohesion was drastically increased, and coupling was decreased since now a subsystem class did not have to call a database class, and retrieval of information was kept within a class. By refactoring the system design in this way, the blob antipattern was avoided since there was no longer a god database class.

For release 2, a session subsystem was added to the design to handle concurrent client requests. This subsystem was separated from the user interface subsystem to decrease coupling. In the system design, the UI subsystem is solely responsible for displaying information, while a specific connection queries for information or makes/deletes reservations. By separating these subsystems, the system allows for there to be multiple client connections.

IV. Subsystems

A. User Interface Subsystem

Subsystem: UI Subsystem
Depends on: all other subsystems
Description: The UI subsystem is responsible for displaying a user interface that allows clients to to communicate with the server. In this release, a graphical user interface has been implemented in addition to the original plain text UI. The subsystem then parses the commands and passes them to the session.
The commands are used to hold the logic for each action that the AFRS needs to perform. These actions range from querying flights to creating reservations. The commands are invoked by the session. This is changed from the first release where the commands were invoked by the UI itself. This was done because of the states of the session. If a session is connected then it has access to all the commands. If the sessions is not connected then it can only access the connect command
Design rationale:

- *Command pattern*: the command pattern allows for commands to be added to a HashMap<String, Command> without having to update the Session implementation to be aware of the new command. This was done to follow Open Close Principle as extension doesn't require massive changes to the Invoker.
- *The Switch to Session as Invoker:* The switch was because of the necessity of multiple connection ids. This new requirement forced a major refactoring of how the commands were called. Previously the commands were called directly from the interface. Now the interface calls method on the session which, depending on the state, then calls the command.

[New in Release 2]

- *Addition of undo and redo:* With the addition of undo and redo, additional commands needed to be created. A command for undoing and for redoing actions were easily created due to the command pattern. To add undo and redo capability, an undo and redo stack was added to the session and methods that call the undo and redo commands. The command pattern made it extremely easy to follow open-close principle.
- *Addition of Connect and Disconnect:* Another new requirement required the ability to connect and disconnect from the server. Again the command structure makes this easy to extend. The disconnect command follows the same path as all other commands. It is invoked by the Session. The connect command is different. The connect command is the only command not invoked by a session because this command creates a new session.
- *Graphical User Interface:* A GUI has been implemented that allows the client to interact with the server in a simpler, more visual way.

- *Input Parser:* This input parser was created to make addition of new commands easier while trying to follow open close principle. By having both UIs transfer the string to another class for parsing, the UIs themselves will not need to be changed for additional commands. This does also increase the cohesion of the system as the UI classes are solely responsible for taking in user input and displaying system response. This does increase the coupling; however, as user input now needs to go through about 5 classes to be done.

GoF Card:

Subsystem name: Commands		**Pattern**: Command
Participants		
Class	**Role**	**Participant's contributions**
Command	Command interface	This is the interface that is implemented by all the concrete commands. This interface only holds the method execute, which executes the logic for each command
ConsoleUI, GUI	Client	The User Interfaces act as the client. The system parses the user input and calls the respective method on the Invoker(client).
Session	Invoker	When the invoker method gets called, it invokes the command via its state(connected or not connected). This then returns the string array to the afrs which displays the returned string
ConnectCommand	Concrete command	This is the only command not invoked by the Session It is instead invoked by the AFRS because this command only creates a new connection.
DisconnectCommand	Concrete command	This command disconnect a connection by removing it from the valid connections. This can only happen in a connected state.
FlightQueryCommand	Concrete command	When created the flight query command also creates the flights in the system. When executed this queries flights from given user input. This command also has a toString which parses the returned list of Itineraries to created the desired output.
CreateReservationCommand	Concrete command	This command implements the logic of creating a reservation. This command gets the flight path from the last query and creates a new reservation and associates it with a passenger name.
RetrieveReservationCommand	Concrete command	This command implements the logic to

		retrieve an existing reservation. It takes in a person's name and possibly an origin and destination and returns the details of the itinerary.
DeleteReservationCommand	Concrete command	This command implements the logic to delete an existing reservation. It deletes a reservation from the current list of created reservations. It takes in the passenger name and the origin and destination
AirportQueryCommand	Concrete command	This command implements the logic to query for airport information. It creates the airport information on creation. When executed it gets the airport information for the inputted airport. This information includes full name, airport weather and delay time
ExitCommand	Concrete command	This command handles the exiting of the program. Although not implemented in release 1 it’s main purpose is to save all reservations to a file.
RedoCommand	Concrete command	This command adds the logic for redoing the sessions last undo. When the Redo command is called the system gets the top item on the redo stack and calls its redo command
UndoCommand	Concrete command	This command adds the logic for undoing the sessions last action. When the Undo command is called the system gets the top item on the undo stack and calls the undo command on that command
Reservation	Receiver	This is the class used to Reservations. It holds the Itinerary for a reservation and the passenger info. This is where created reservations get stored and this is where the get deleted from.
AirportInfo	Receiver	This is the class that represents the airport info. The airport info command comes here to get the airport info.
FlightQueryCommand	Receiver	This is the class that creates itineraries based of the user input. The flight query

		command calls the create itinerary method to get the itineraries for the requested path.
Deviations from the pattern	The invoker isn’t the same for all commands. Every command but 1 share the same invoker. The Connect command’s invoker is the afrs system	
Requirements fulfilled by this pattern	Requirement 1	

UML Diagram:

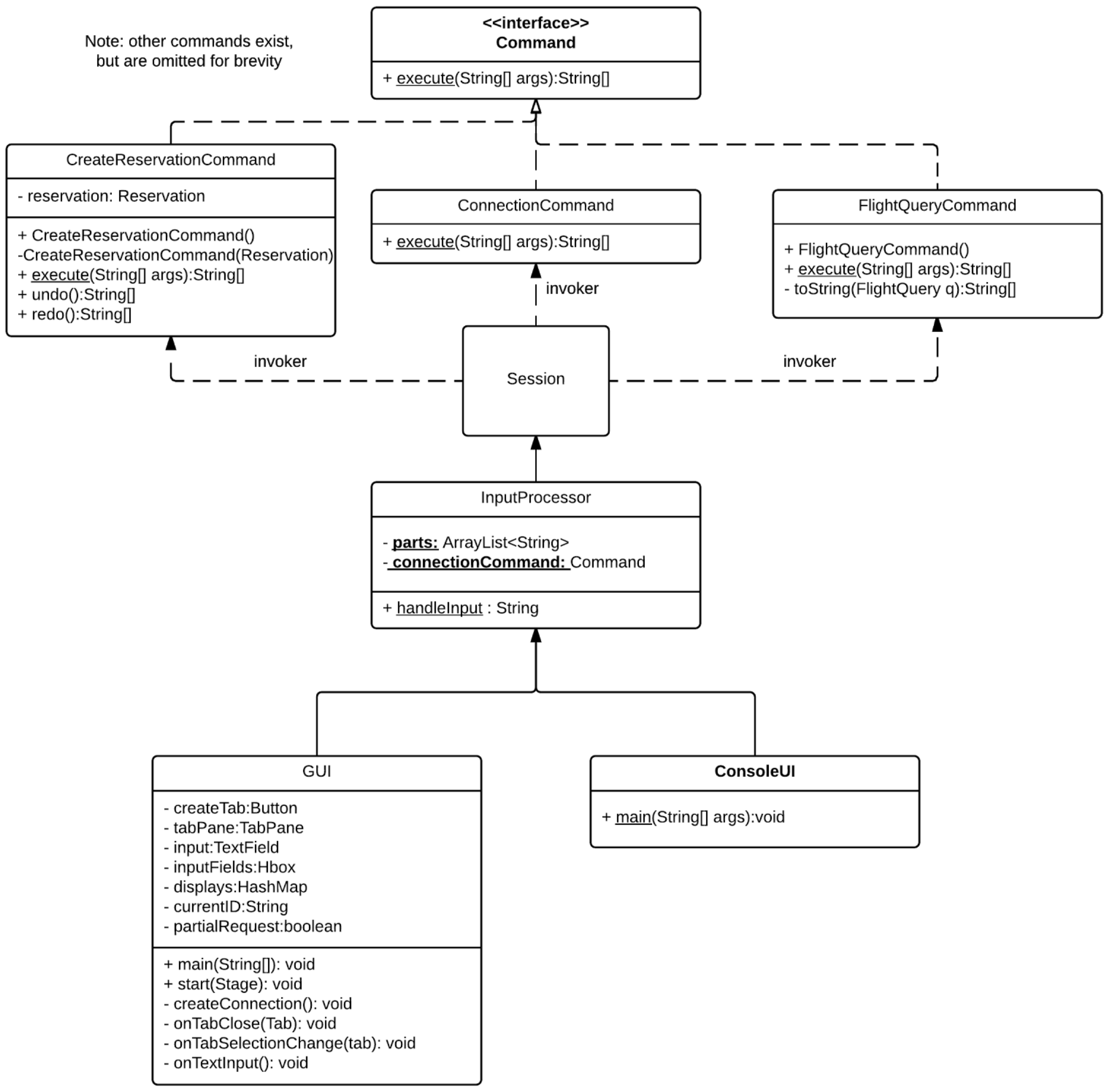

Note: other commands exist, but are omitted for brevity
<<interface>> Command
+ execute(String[] args):String[]
CreateReservationCommand
- reservation: Reservation
+ CreateReservationCommand()
-CreateReservationCommand(Reservation)
+ execute(String[] args):String[]
+ undo():String[]
+ redo():String[]
ConnectionCommand
+ execute(String[] args):String[]
FlightQueryCommand
+ FlightQueryCommand()
+ execute(String[] args):String[]
- toString(FlightQuery q):String[]
invoker
invoker
invoker
Session
InputProcessor
- parts: ArrayList<String>
- connectionCommand: Command
+ handleInput : String
GUI
- createTab:Button
- tabPane:TabPane
- input:TextField
- inputFields:Hbox
- displays:HashMap
- currentID:String
- partialRequest:boolean
+ main(String[]): void
+ start(Stage): void
- createConnection(): void
- onTabClose(Tab): void
- onTabSelectionChange(tab): void
- onTextInput(): void
ConsoleUI
+ main(String[] args):void

Figure 4A-01: Command UML

Sequence Diagram:

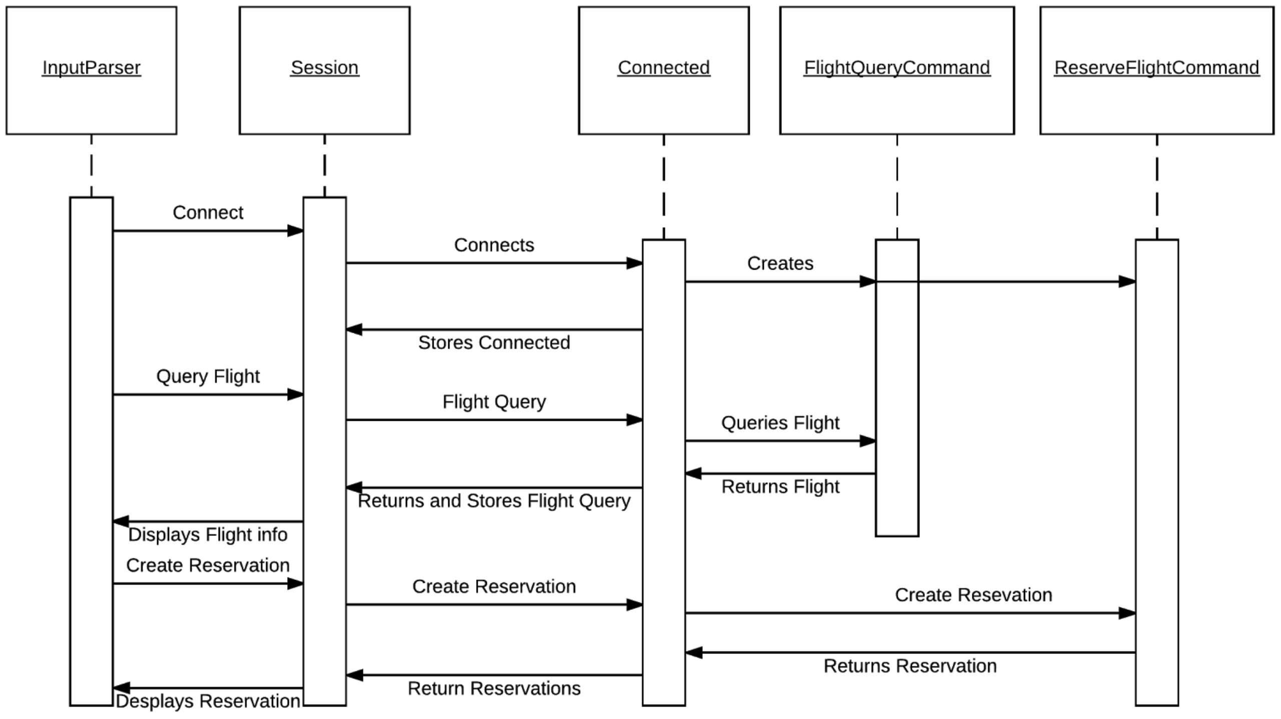


Figure 4A-02: Sequence Diagram for querying and creating reservations through the new connection state.

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. The system will provide the option to query airport information from a local test file or get real-time information from the Federal Aviation Administration (FAA) web service. 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: To query for airport information, the client shall enter in the text-string: "airport, " followed by the three-letter code for the airport that the user requests and then send the request with a "; " in the user interface. The user will also be able to choose whether to query the information from a local test file or the Federal Aviation Administration (FAA) web service.

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. The system will ask the user the information source he/she would like to use.
3. AirportQueryCommand will then call getAirportInfo, to get the airport string that corresponds to the airport code if the user chooses local test file.
4. AirportInfo reads in a file of airport codes and corresponding airport names, a file with a set of weather and temperatures for each airport code, and a file with airport codes and the corresponding delay times.
5. After parsing through the file, the system stores the information [name, arraylist of weather and temperatures, and delay] associated with each airport code into a hashmap with the airport code as the key value.
6. If the user chooses to query for data from the FAA web service, the AirportQueryCommand will call request() to request access from the remote proxy which will pull and parse from the XML file to get the airport name, weather, temperature, and average delay time. **[New in Release 2]**
7. Finally, the information, will be displayed in the following format: (airport, [airportName], [airportWeather], [airportTemp], [airportCurrent Delay])

   airport, Orlando, sunny, 95, 15

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

Design Rationale:

- *AirportInfo class:* The AirportInfo class, in Release 1, consisted of many inefficiencies that were redundant and time-consuming. Firstly, to check the validity of an airport code, this class parsed through an entire file to check to see if the airport code had an airport name associated with it each time the system called the isValid method. In this release, this method was changed to search a hashmap that was created at the initiation of the system for the key (airport code). The code to create a hashmap containing all of the airport information was moved to the constructor of the AirportInfo class so that the hashmap would read in the files and store the information once, instead of each time the user queried for an airport code. The AirportInfo class can also request to get airport information from the FAA web service, which is then redirected to the proxy class.

- *HashMap over ArrayList:* Since there were three separate files to hold the airport information, we initially were contemplating to make three separate ArrayLists. This concept would have adhered to the separation of concerns principle since any other class using airport information would not have to go through a conglomerated ArrayList. After implementing the three different ArrayLists, we saw that it was difficult for a class to know which ArrayList to call without knowing the code from AirportInfo; this is a violation of the Law of Demeter. As a result, a Hashmap was used to store the airport information with the airport code as the key.

[New in Release 2]

- *Remote Proxy Pattern:* In Release 1, the Observer Pattern was shoehorned to fit the Airport Information subsystem, when there did not need to be a design pattern to fit such a straightforward implementation. In Release 2, the Remote Proxy pattern was suitably chosen to handle the new requirement of FAA web service requests. The proxy class used in the design pattern provides a surrogate or placeholder to provide access to an object. In the airport query subsystem, the proxy class is the FAAairportInfo class, which provides a level of indirection for when the client requests airport information. In this Representational State Transfer proxy (REST) the proxy forwards the request method call to the FAA web service, which resides on a different machine than that of the client. This pattern also allows for higher security measures, in case the connection is unsuccessful.

- *FAAairportInfo class:* This class was added to the implementation from Release 1 to function as a RESTful proxy to get airport information from the FAA web service. This class maintains a reference that allows the Proxy to access the RealSubject. Since both AirportInfo and FAAairportInfo implement the AirportProxy interface, the Proxy can be substituted for the RealSubject.

- *Use of Strategy Pattern:* The strategy pattern was used to determine what server to use. Whether using the local server or the faa server the strategy pattern made switching between these two extremely easy. It also added the ability to expand the system while still following Open Close Principle.

Proxy Pattern:

GOF Card:

Subsystem Name: Airport Information		**GoF pattern:** Proxy Pattern
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
AirportQueryCommand	Client	AirportQueryCommand is the client that calls the request() method from AirportInfo [RealSubject] to request the airport information from the FAA web service.
AirportProxy : Interface	Subject	The AirportProxy interface is implemented by AirportInfo [RealSubject] and FAAairportInfo [Proxy] and represents the request() service.
FAAairportInfo	Proxy	This class functions as a RESTful proxy to get airport information from the FAA web service. It maintains a reference that allows the Proxy to access the RealSubject. Since both AirportInfo and FAAairportInfo implement the AirportProxy interface, the Proxy can be substituted for the RealSubject. The FAAairportInfo class, in a way, controls access to AirportInfo [RealSubject] and is responsible for the creation or deletion of a request to the FAA web service.
AirportInfo	RealSubject	The AirportInfo class calls the Proxy class (FAAairportInfo) to get the FAA web service information. AirportInfo is the real object that the proxy is representing.
Deviations from the standard pattern: N/A		
Requirements being covered: Functional Requirement → 4i and 4ii Non-Functional Requirement → 4a, 4b, 4c, and 4d		

UML Diagram:

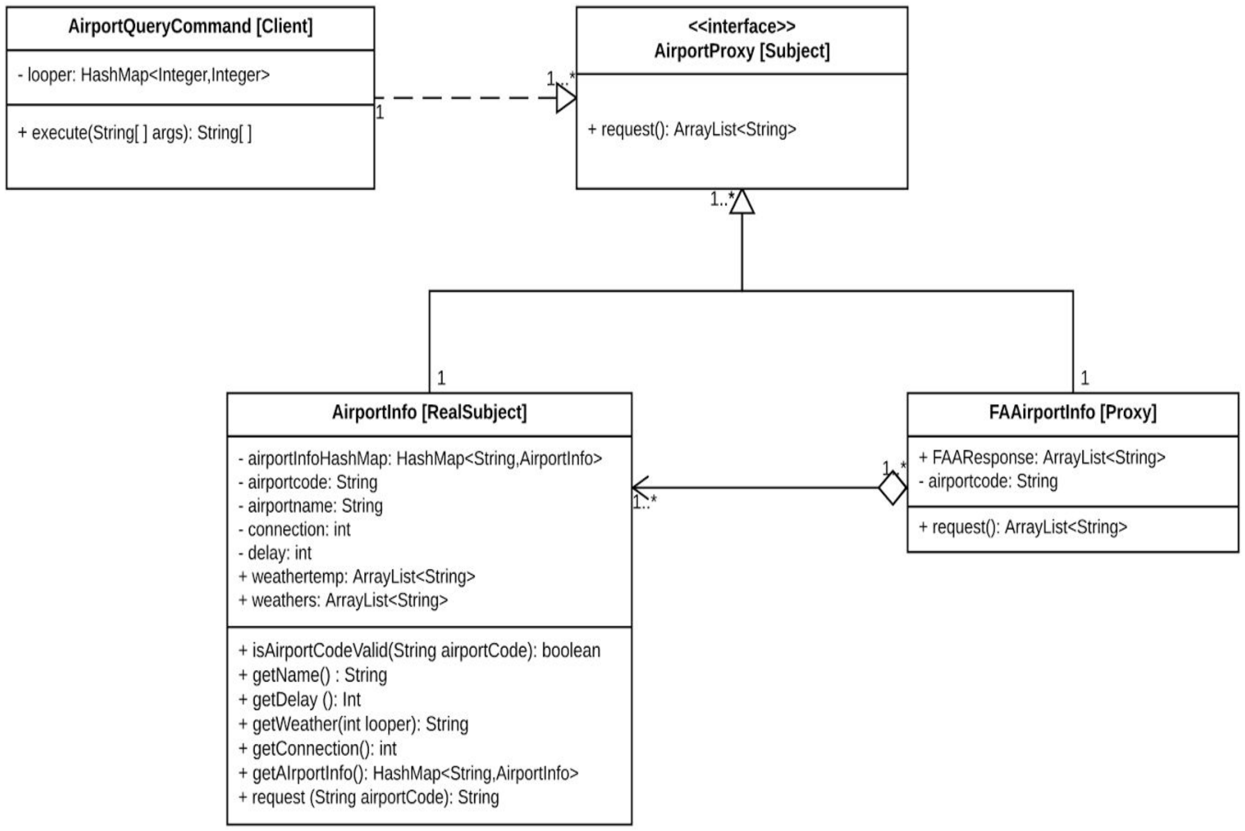

AirportQueryCommand [Client]
- looper: HashMap<Integer,Integer>
+ execute(String[ ] args): String[ ]
1
1..*
<<interface>>
AirportProxy [Subject]
+ request(): ArrayList<String>
1..*
1
1
AirportInfo [RealSubject]
- airportInfoHashMap: HashMap<String,AirportInfo>
- airportcode: String
- airportname: String
- connection: int
- delay: int
+ weathertemp: ArrayList<String>
+ weathers: ArrayList<String>
+ isAirportCodeValid(String airportCode): boolean
+ getName() : String
+ getDelay (): Int
+ getWeather(int looper): String
+ getConnection(): int
+ getAIrportInfo(): HashMap<String,AirportInfo>
+ request (String airportCode): String
1..*
1..*
FAAirportInfo [Proxy]
+ FAAResponse: ArrayList<String>
- airportcode: String
+ request(): ArrayList<String>

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

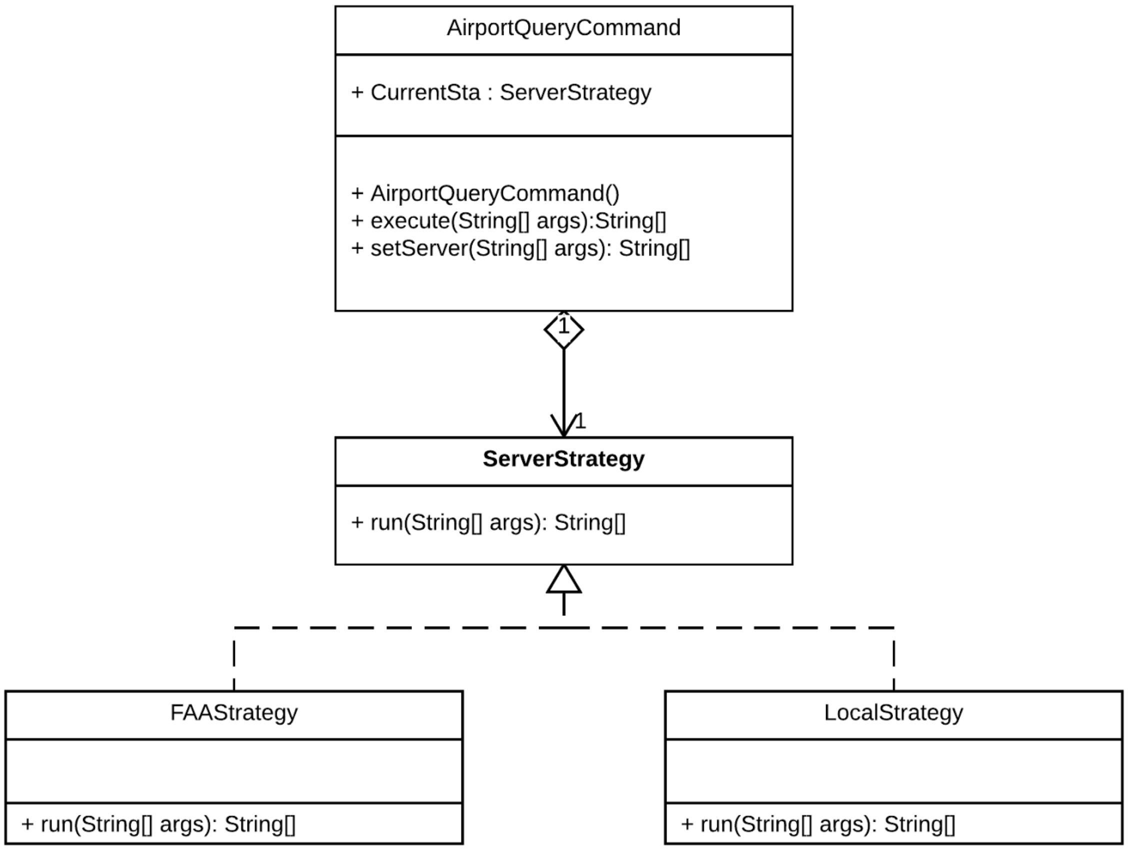
UML Diagram Note:

The AirportQueryCommand is part of both the Command Pattern and the Proxy Pattern. The user interface subsystem implements the command system to carry out each subsystem; the AirportQueryCommand serves as a concrete command for this design pattern. In the proxy pattern, the AirportQueryCommand is the client that calls AirportInfo which makes the FAA service request to the proxy.

Strategy Pattern:

Subsystem Name: Airport Information		**GoF pattern:** Strategy Pattern
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ServerStrategy: Interface	Strategy	The ServerStrategy interface defines an interface common to all supported algorithms (FAA strategy or local strategy). The AirportQueryCommand uses this interface to call the run() algorithm defined by either ConcreteStrategy.
AirportQueryCommand	Context	The AirportQueryCommand class contains a reference to the Concrete Strategy (FAAStrategy or LocalStrategy) that should be used. When a user sets a server through a command then the algorithm is run from the strategy object.
FAAStrategy	Concrete Strategy [Implementation One]	This class implements the algorithm to get information from the FAA server. This represents, one of the services a user can use in order to obtain airport information.
LocalStrategy	Concrete Strategy [Implementation Two]	This class implements the algorithm to get information from the local test files. This represents, one of the services a user can use in order to obtain airport information.
Deviations from the standard pattern: N/A		
Requirements being covered: Non-Functional Requirement → 4a and 4d		

UML Diagram:


AirportQueryCommand
+ CurrentSta : ServerStrategy
+ AirportQueryCommand()
+ execute(String[] args):String[]
+ setServer(String[] args): String[]
1
1
ServerStrategy
+ run(String[] args): String[]
FAAStrategy
+ run(String[] args): String[]
LocalStrategy
+ run(String[] args): String[]

UML Diagram Note: While still maintaining the integrity of the Proxy pattern that is used in the Airport Information subsystem, the Strategy pattern was also implemented in order to represent the two servers that a user can switch between. The AirportQueryCommand was also used as the context for the Strategy pattern applied to this subsystem. Depending on the user input of what server they wish to use, then that particular run() method will be executed.

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 system to create Itineraries based on the queried flights. Both the itineraries and flightInfo classes implement the itinerary interface. This design 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 is not necessarily used in this system, it does mean that an Itinerary can hold other Itineraries. Along with the standard 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 is held in the Itinerary. We also have created a flights class that on startup is created and creates all the flight information. This class then holds a list of all flights.

Design Rationale:

- *Composite pattern:* The composite pattern was chosen because it allowed for the creation of different lengths of 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 held over how to hold flight information in the system. The first was to create all flights on startup and hold them in the system. Another idea was to create all itineraries at the start of the system and just query the already created itineraries. The final was to create flights as they were queried. The first method was chosen since creating all itineraries at initiation 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. It was decided that the system should create all flightInfo objects on startup and store them. The Flight Class creates them on initiation and saves them in a state to 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 share. They share the getOrign, getDestination, getAirfare, getArrivalTime, and getDepartureTime methods.
FlightInfo	Leaf	The Flight class represents the most simple element of an itinerary and does not have any children. This holds the arrival time, departure time, arrival airport, destination airport, flight number, and cost.
Itinerary	Composite	The Itinerary class represents an element that has children whether it is flights or itineraries themselves. The itinerary hold the destination airport, arrival airport, list of flights and an ID number. The itinerary can also calculate the arrival time, departure time, and airfare from the list of flights
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:

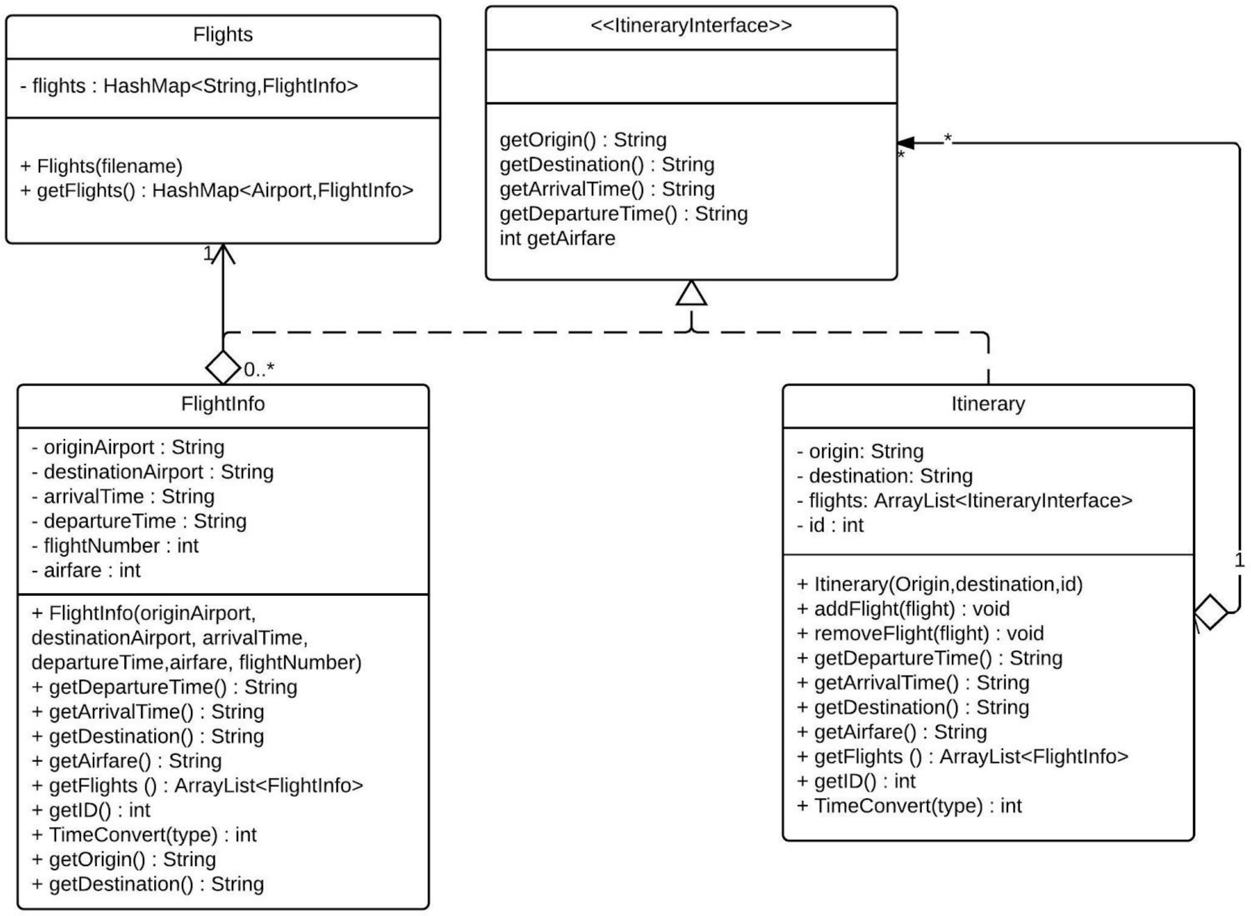

Flights
- flights : HashMap<String,FlightInfo>
+ Flights(filename)
+ getFlights() : HashMap<Airport,FlightInfo>
<<ItineraryInterface>>
getOrigin() : String
getDestination() : String
getArrivalTime() : String
getDepartureTime() : String
int getAirfare
FlightInfo
- originAirport : String
- destinationAirport : String
- arrivalTime : String
- departureTime : String
- flightNumber : int
- airfare : int
+ FlightInfo(originAirport, destinationAirport, arrivalTime, departureTime,airfare, flightNumber)
+ getDepartureTime() : String
+ getArrivalTime() : String
+ getDestination() : String
+ getAirfare() : String
+ getFlights () : ArrayList<FlightInfo>
+ getID() : int
+ TimeConvert(type) : int
+ getOrigin() : String
+ getDestination() : String
Itinerary
- origin: String
- destination: String
- flights: ArrayList<ItineraryInterface>
- id : int
+ Itinerary(Origin,destination,id)
+ addFlight(flight) : void
+ removeFlight(flight) : void
+ getDepartureTime() : String
+ getArrivalTime() : String
+ getDestination() : String
+ getAirfare() : String
+ getFlights () : ArrayList<FlightInfo>
+ getID() : int
+ TimeConvert(type) : int
1
0..*
*
*
1

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

Sequence Diagram:

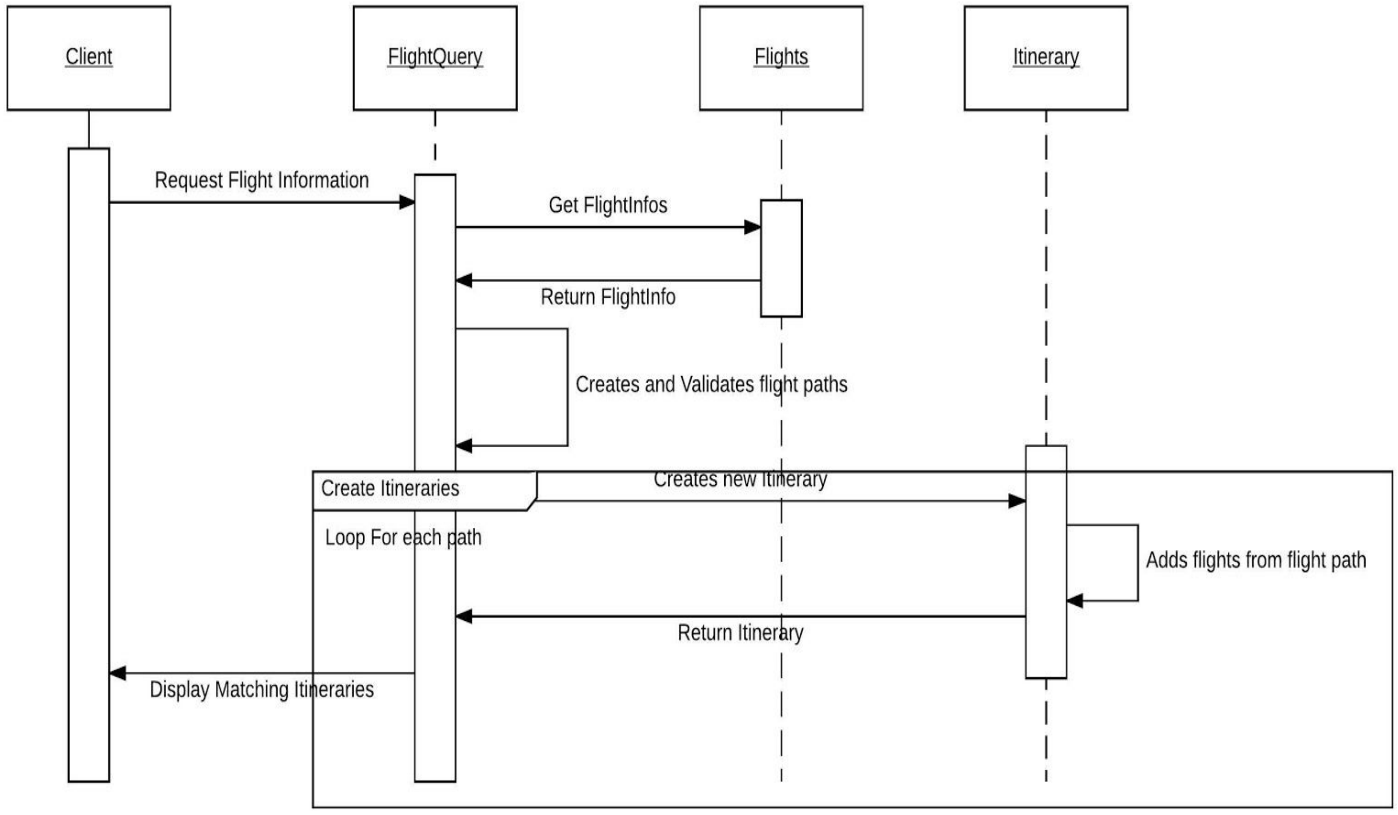


Figure 4.C-02: Sequence Diagram representing the Itinerary and Flight Query SubSystem

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 takes in a users request for flights and creates 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, they are added to the success list, if not they are added to the path list. This process continues until the max number of connections are reached, and all flights in the success list get validated for a 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 flight 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, the strategy pattern would be appropriate to use.

- *Hashmap over ArrayList:* Originally, an ArrayList was used to store all the flights. The system had performance problems when finding all possible paths out of the airports since the system was iterating over an ArrayList over and over again. To fix the performance issues, a hashmap was implemented instead, where the keys were the airport codes, and the values were the flights.

- *isValid:* In the initial design, each path would be validated as it was created, but this proved too difficult in the time available, so an isValid method was implemented into the system. This method checks the whole path to see if the times work out and to see if the flight end in the origin. A performance issue occurred with this design since this method was checking so many flight paths.

- *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 the list is returned instead of the all possible path list. This significantly cuts 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 (i.e., 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. Its main job is to find and validate all paths from origin to destination. Then using those valid paths creates itineraries. 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. It takes in an arrayList of itineraries and sorts them based on departure time.
ArivalTime	ConcreteStrategy B	This holds the sorting algorithm for sorting by Arrival Time. It takes in an arrayList of itineraries and sorts them based on arrival time.
Airfare	ConcreteStrategy C	This holds the sorting algorithm for sorting by Airfare It takes in an arrayList of itineraries and sorts them based on airfare.
Deviations from the standard pattern:		
Requirements being covered: 2B, 2C, 2D, 2E, 2F		

UML Diagram:

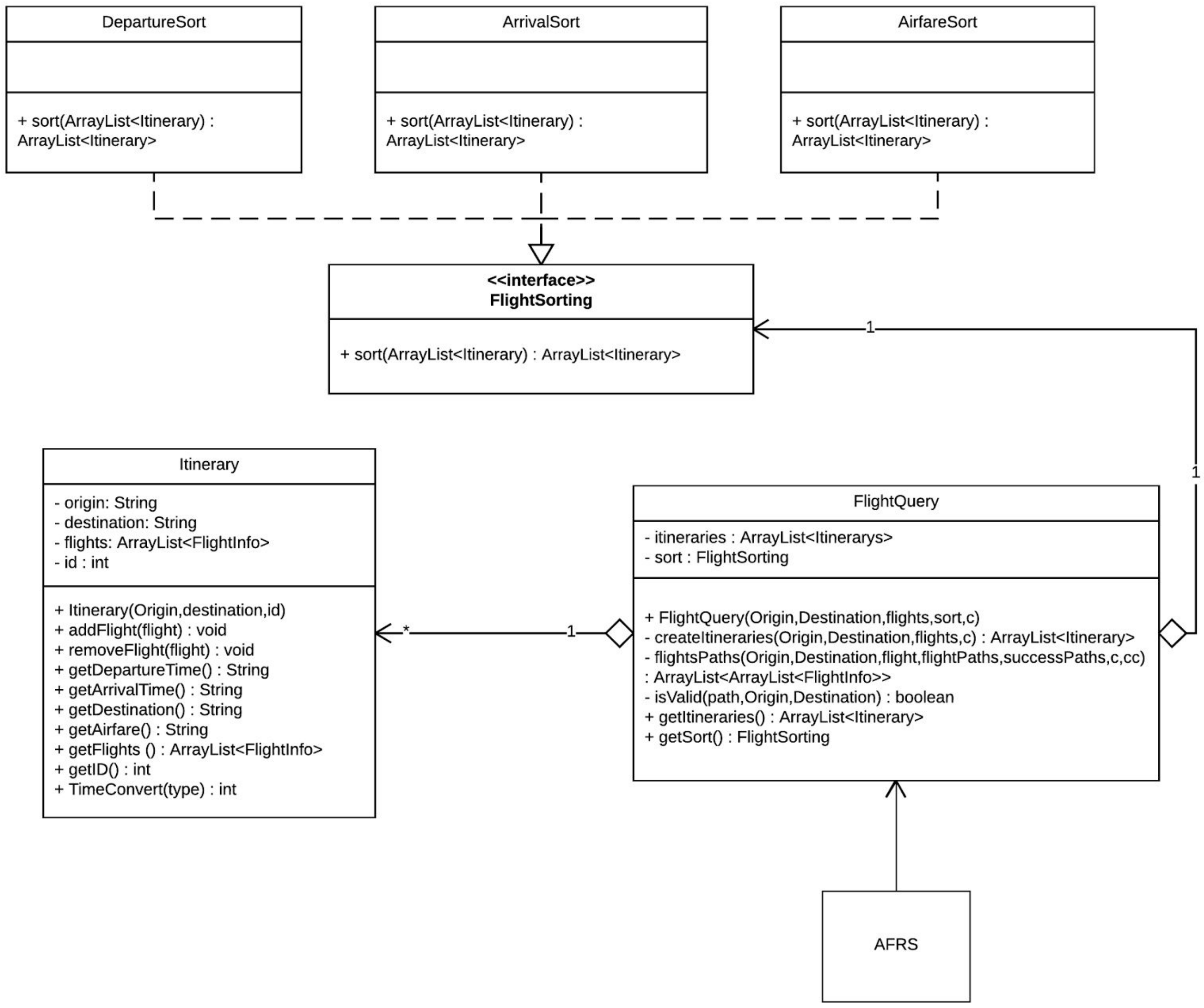

DepartureSort
+ sort(ArrayList<Itinerary) : ArrayList<Itinerary>
ArrivalSort
+ sort(ArrayList<Itinerary) : ArrayList<Itinerary>
AirfareSort
+ sort(ArrayList<Itinerary) : ArrayList<Itinerary>
<<interface>>
FlightSorting
+ sort(ArrayList<Itinerary) : ArrayList<Itinerary>
1
1
Itinerary
- origin: String
- destination: String
- flights: ArrayList<FlightInfo>
- id : int
+ Itinerary(Origin,destination,id)
+ addFlight(flight) : void
+ removeFlight(flight) : void
+ getDepartureTime() : String
+ getArrivalTime() : String
+ getDestination() : String
+ getAirfare() : String
+ getFlights () : ArrayList<FlightInfo>
+ getID() : int
+ TimeConvert(type) : int
*
1
FlightQuery
- itineraries : ArrayList<Itinerarys>
- sort : FlightSorting
+ FlightQuery(Origin,Destination,flights,sort,c)
- createItineraries(Origin,Destination,flights,c) : ArrayList<Itinerary>
- flightsPaths(Origin,Destination,flight,flightPaths,successPaths,c,cc)
: ArrayList<ArrayList<FlightInfo>>
- isValid(path,Origin,Destination) : boolean
+ getItineraries() : ArrayList<Itinerary>
+ getSort() : FlightSorting
AFRS

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

Sequence Diagram:

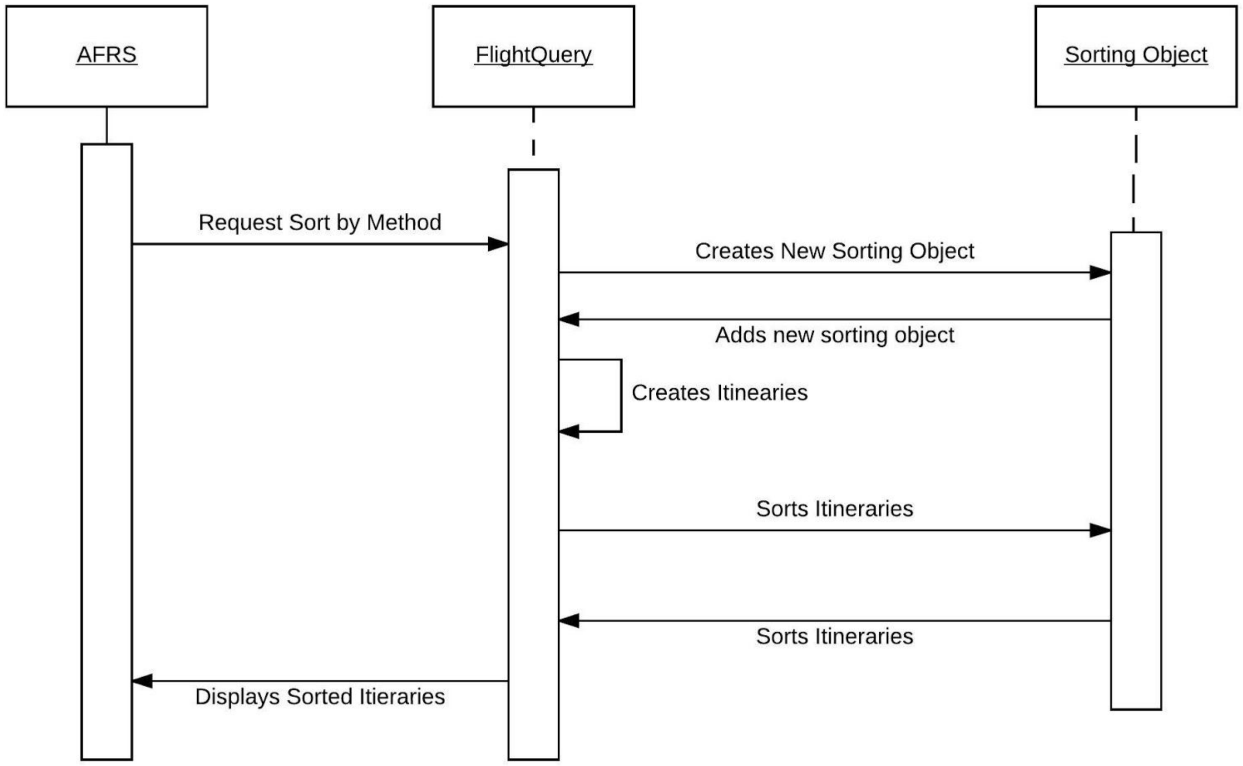


Figure 4.D-02 : Sequence Diagram for sorting Flights

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:

- *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.

[New in Release 2]

- *Persistence*: the Reservation subsystem has the ability to save reservations upon session disconnect, and load those reservations upon application startup. The JSON format is used as libraries are widely available, and it is easy to manually verify correct serialization.
- *Undo/redo*: the commands for creating and deleting reservations support undo and redo, they also contain a field to hold onto the reservation they will undo/redo. The actual command instances are held on stacks in Session class, which actually calls the undo/redo methods on the commands.

UML diagram:

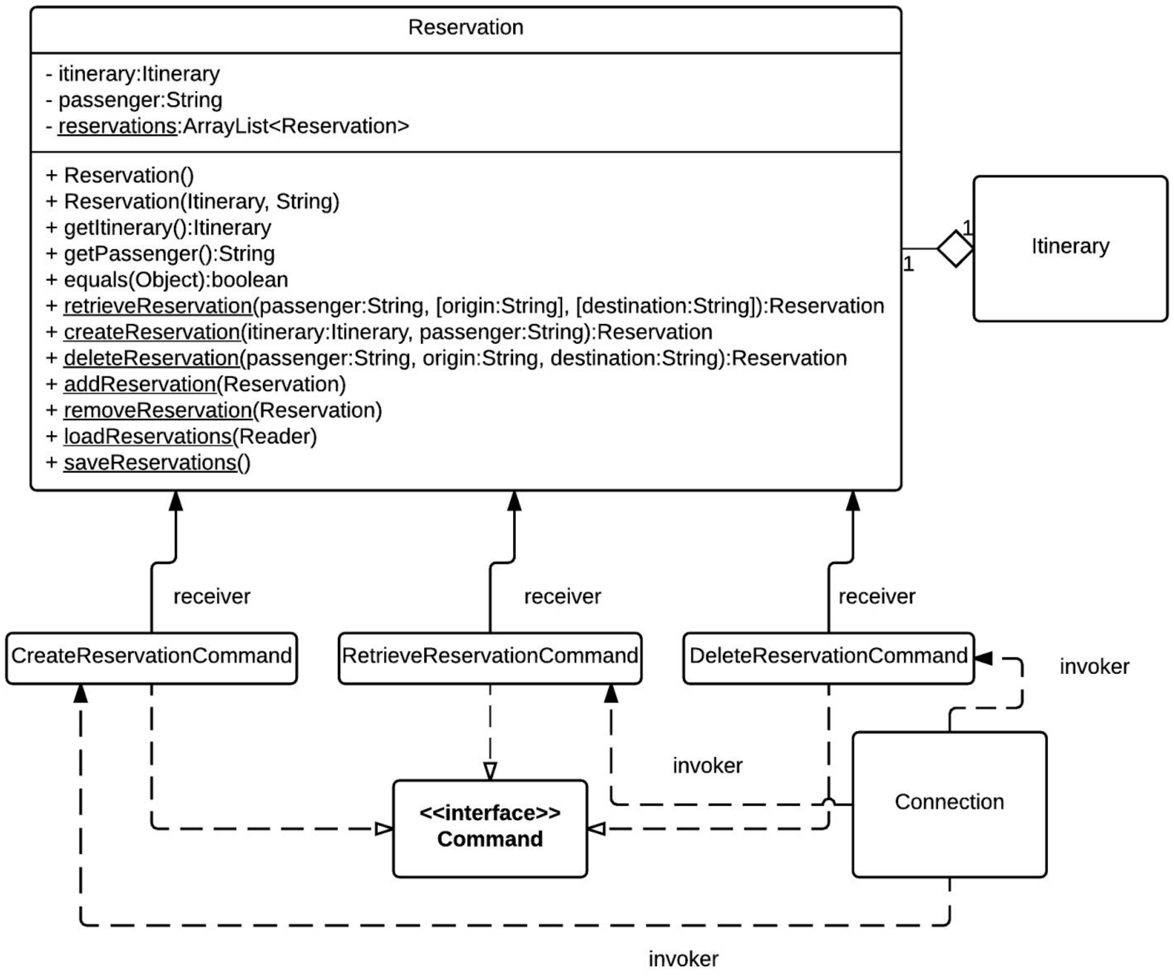

Reservation
- itinerary:Itinerary
- passenger:String
- reservations:ArrayList<Reservation>
+ Reservation()
+ Reservation(Itinerary, String)
+ getItinerary():Itinerary
+ getPassenger():String
+ equals(Object):boolean
+ retrieveReservation(passenger:String, [origin:String], [destination:String]):Reservation
+ createReservation(itinerary:Itinerary, passenger:String):Reservation
+ deleteReservation(passenger:String, origin:String, destination:String):Reservation
+ addReservation(Reservation)
+ removeReservation(Reservation)
+ loadReservations(Reader)
+ saveReservations()
1
1
Itinerary
receiver
receiver
receiver
CreateReservationCommand
RetrieveReservationCommand
DeleteReservationCommand
invoker
invoker
<<interface>>
Command
Connection
invoker

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

E. Session Subsystem:

Subsystem: The Session Subsystem

Requirements: This subsystem allows for and keeps track of the different connections that the AFRS must handle. This subsystem allows for flight queries and undo/redo stacks to be unique to a connection

Description: This subsystem consist of 4 classes. The most important class is the Session class. This class represents a connection is and is created by the connection command. Here the session class holds the connection id, last queried flights, the looper for local airport information, and the current state. This subsystem does use the state design pattern. The two states for a session are connected and not connected. When flight info is queried, the UI calls flight query on the connection. This connection then calls the same method on the current state. If the session is not connected all methods return an error. If a session is connected, the methods call the commands associated with the action. This class has replaced the AFRS class as the Invoker of commands as well

Design Rationale:

- *Use of State Pattern* - Here state pattern was used because of the two states of a session. When the session is not connected, the session can not do anything. When the session is connected, the session can act as normal. These two states change the behavior of the session greatly.
- *Session class over Hashmap* - Originally a hash map would have been used to keep track of different sessions flight queries and undo redo stacks. This was what was originally implemented but changed as the development when on. Creation of a Session class was used mostly because of the two states of connection. The original hashmap idea would not have been well equipt to handle this and thus needed to be changed. A large refactoring was done to create the session class and use state pattern.

GoF Card:

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
UI	Client	The Client is the UI. The UI parses the user inputs and calls the respective methods on the context object the Session. The UI also gets back the string array response that action provides and displays it.
Session	Context	The session stores information related to a specific session like connection id or last flight query. It has methods for querying flights, creating deleting or retrieving reservations, and retrieving airport information. These methods call the respective method on the current state, which can be connected or not connected.
State	State	This is the interface for the states. The methods each state have are flight query, retrieve, delete, or create reservation, get airport information, switch airport informations server, and disconnect. All these methods return a String array which is the desired output for the UI with each action
Connected	Concrete State	The Connected State is the set of behaviors for if the session is connected. If the session is connected each method class the respective execute on the command. This allows the system to behave the same as the first release. These methods also return the String array that the commands produce. The Connected State also holds a command array for the session
NotConnected	Concrete State	The Not Connected State is the set of behaviors for if the system is not connected. Instead of calling the commands like the connected state, this state only returns a string array. This string array only has one element which is “error,invalid connection”.
Deviations from the standard pattern: N/A		
Requirements being covered: Requirement 2		

State Diagram:

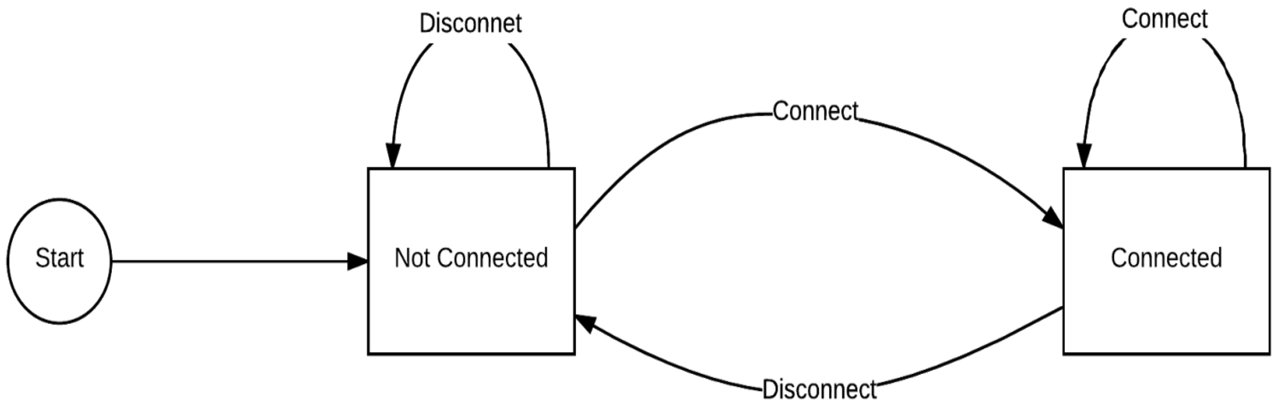


Figure 4E-01: The State Diagram for session

UML Diagram:

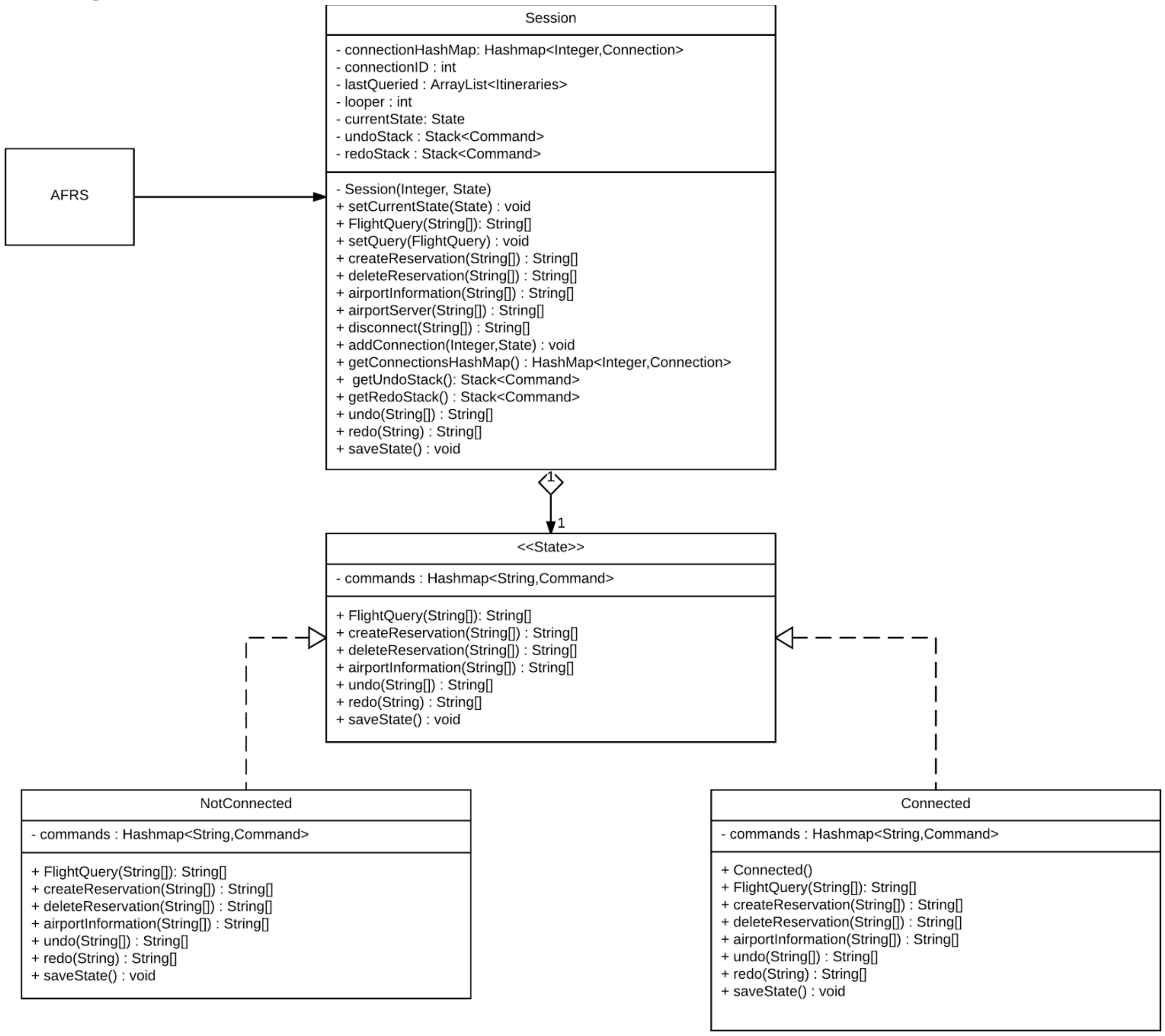

Session
- connectionHashMap: Hashmap<Integer,Connection>
- connectionID : int
- lastQueried : ArrayList<Itineraries>
- looper : int
- currentState: State
- undoStack : Stack<Command>
- redoStack : Stack<Command>
- Session(Integer, State)
+ setCurrentState(State) : void
+ FlightQuery(String[]): String[]
+ setQuery(FlightQuery) : void
+ createReservation(String[]) : String[]
+ deleteReservation(String[]) : String[]
+ airportInformation(String[]) : String[]
+ airportServer(String[]) : String[]
+ disconnect(String[]) : String[]
+ addConnection(Integer,State) : void
+ getConnectionsHashMap() : HashMap<Integer,Connection>
+ getUndoStack(): Stack<Command>
+ getRedoStack() : Stack<Command>
+ undo(String[]) : String[]
+ redo(String) : String[]
+ saveState() : void
AFRS
1
1
<<State>>
- commands : Hashmap<String,Command>
+ FlightQuery(String[]): String[]
+ createReservation(String[]) : String[]
+ deleteReservation(String[]) : String[]
+ airportInformation(String[]) : String[]
+ undo(String[]) : String[]
+ redo(String) : String[]
+ saveState() : void
NotConnected
- commands : Hashmap<String,Command>
+ FlightQuery(String[]): String[]
+ createReservation(String[]) : String[]
+ deleteReservation(String[]) : String[]
+ airportInformation(String[]) : String[]
+ undo(String[]) : String[]
+ redo(String) : String[]
+ saveState() : void
Connected
- commands : Hashmap<String,Command>
+ Connected()
+ FlightQuery(String[]): String[]
+ createReservation(String[]) : String[]
+ deleteReservation(String[]) : String[]
+ airportInformation(String[]) : String[]
+ undo(String[]) : String[]
+ redo(String) : String[]
+ saveState() : void

Figure 4E-02 : UML for Session's State Pattern

Status of the Implementation:

User Interface Subsystem:

- Bug with closing connect and reconnecting lead to a disconnected tab to be created. The next connection works however.
- Reservations are only persisted on disconnects of session and not closure of window
- Exit command exist but does not work. It is not used the release 2.

Airport Information Subsystem:

- A FAA server option was added to allow the user to choose between using the local test files or the FAA server
- Sometimes the FAA server option will return no delay time. This is most likely that the FAA web server does not have the information available

Flight Query Subsystem:

- Fixed performance issues experienced with release one.

Reservation Subsystem:

- Reservations are now persisted, unlike R1

Session Subsystem:

- Works as intended however can be considered a blob

Appendix

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 reads in three files for the local test service and makes a hashmap with the airport code as the key and the value contains airport name, weather, temperature, and delay time. Also has the functionality to check to see if an airport code is valid. The AirportInfo class also acts as a RealSubject for the FAAairportInfo proxy; there is a request() method to request the FAA server information from the proxy.	
Collaborators	
Uses: AirportProxy, FAAairportInfo	**Used by:** AirportQueryCommand
Author: Niharika Reddy	

Class: AirportQueryCommand	
Responsibilities: This class is a command in the command pattern for the airport query. It acts as a concrete Command in the command pattern. If a user chooses to set the server to faa, then calls the request() method from AirportInfo.	
Collaborators	
Uses: AirportInfo	**Used by:** Command
Author: Niharika Reddy	

Class: AirportProxy	
Responsibilities: This interface is the Subject in the proxy pattern and is implemented by both AirportInfo and FAAairportInfo. It allows for the creation of a remote proxy to retrieve information from the FAA server.	
Collaborators	
Uses:	**Used by:** FAAairportInfo, AirportInfo
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: Connection	
Responsibilities: Represents the connected state for the Session	
Collaborators:	
Uses: State	**Used by:** Session
Author: Stephen Cook	

Class: ConnectCommand	
Responsibilities: Holds the Logic for creating a new conenction	
Collaborators:	
Uses: Conmmand	**Used by:** Session
Author: Stephen Cook	

Class: ConsoleUI	
Responsibilities: Implements the program entry point and text user interface	
Collaborators:	
Uses: InputProcessor	**Used by:**
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: DisconnectCommand	
Responsibilities: Represents the logic for disconnecting from a server	
Collaborators:	
Uses: Command	**Used by:** Session
Author: Stephen Cook	

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

Class: FAAairportInfo	
Responsibilities: This class functions as a RESTful proxy to get airport information from the FAA web service.It maintains a reference that allows the Proxy to access the RealSubject. Since both AirportInfo and FAAairportInfo implement the AirportProxy interface, the Proxy can be substituted for the RealSubject. The FAAairportInfo class, in a way, controls access to AirportInfo [RealSubject] and is responsible for the creation or deletion of a request to the FAA web service. This class pulls from the FAA web service and converts the XML file into strings to store in an array.	
Collaborators:	
Uses: AirportProxy	**Used by:** AirportInfo
Author: Niharika Reddy	

Class: FAAStrategy	
Responsibilities: This class is a command in the command pattern for the airport query acts as a concCommand in the command pattern	
Collaborators:	
Uses: ServerStrategy	**Used by:** AirportQuery
Author: Niharika Reddy	

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: GUI	
Responsibilities: Creates a graphical user interface allowing the creation of new connections, and permitting the client to interact with the server via commands. Also implements the primary entry point for the program.	
Collaborators:	
Uses: InputProcessor	**Used by:**
Author: Joshua Cotton, Moisés Lora	

Class: InputProcessor	
Responsibilities: Processes input given to it by the GUI or TUI and returns a response	
Collaborators:	
Uses: Session, Command, ConnectCommand	**Used by:** GUI, ConsoleUI
Author: Joshua Cotton	

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: LocalStrategy	
Responsibilities: This defines the method for the local server strategy..	
Collaborators:	
Uses: ServerStrategy	**Used by:** AirportQueryCommand
Author: Stephen Cook	

Class: NotConnected	
Responsibilities: This acts as one of the states for a session. When a session isn't connected all the logic is locked to it.	
Collaborators:	
Uses: State	**Used by:**Session
Author: Stephen Cook	

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

Class: RedoCommand	
Responsibilities: Redoes the actions of another command that pushed itself onto the redo stack.	
Collaborators:	
Uses: Command, Reservation	**Used by**: Session
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**: Session
Author: Joshua Cotton	

Class: ServerCommand	
Responsibilities: Command to allow the user to set the server to either local or faa. On initialization, the server is automatically configured to local, until it is changed by the user.	
Collaborators:	
Uses: Command	**Used by**: Session
Author: Niharika Reddy	

Class: Session	
Responsibilities: This class is used to represent an individual session. It hold the states, last queried. undoStack and redoStack.	
Collaborators:	
Uses: State	**Used by:** Command
Author: Stephen Cook	

Class: ServerStrategy	
Responsibilities: The interface used by the concrete strategy	
Collaborators:	
Uses:	**Used by:** FaaServer, LocalServer
Author: Stephen Cook	

Class: State	
Responsibilities: This is the interface used by the concrete states. It defines the methods for Flight Query, Reservation methods, Airport methods and undo redo commands	
Collaborators:	
Uses:	**Used by:** Connected NotConnected State
Author: Stephen Cook	

Class: UndoCommand	
Responsibilities: Command that undoes the actions of another command that pushed itself onto the undo stack.	
Collaborators:	
Uses: Command, Reservation	**Used by**: Session
Author: Joshua Cotton	