SWEN 262

AFRS RELEASE 2

November 8, 2017

Devin Matte drm8019@rit.edu

Amanda Ramos alr7924@rit.edu

Nicholas Montemorano nmm9422@rit.edu

Oren Rosin oxr3231@rit.edu

Contents

SUMMARY

The AFRS lets clients request reservations for flights. The system responds with the possible flight plans, giving the user the power to choose one. The system will ask the user for a command and said user can put a command in to make, delete or save a reservation for a passenger name. The system keeps track of all reservations and flight data. While in the terminal, the user can also input a code for an airport to view all information of the airport such as weather, delay times, flights and location. A user can also get date for each flight stored in the system.

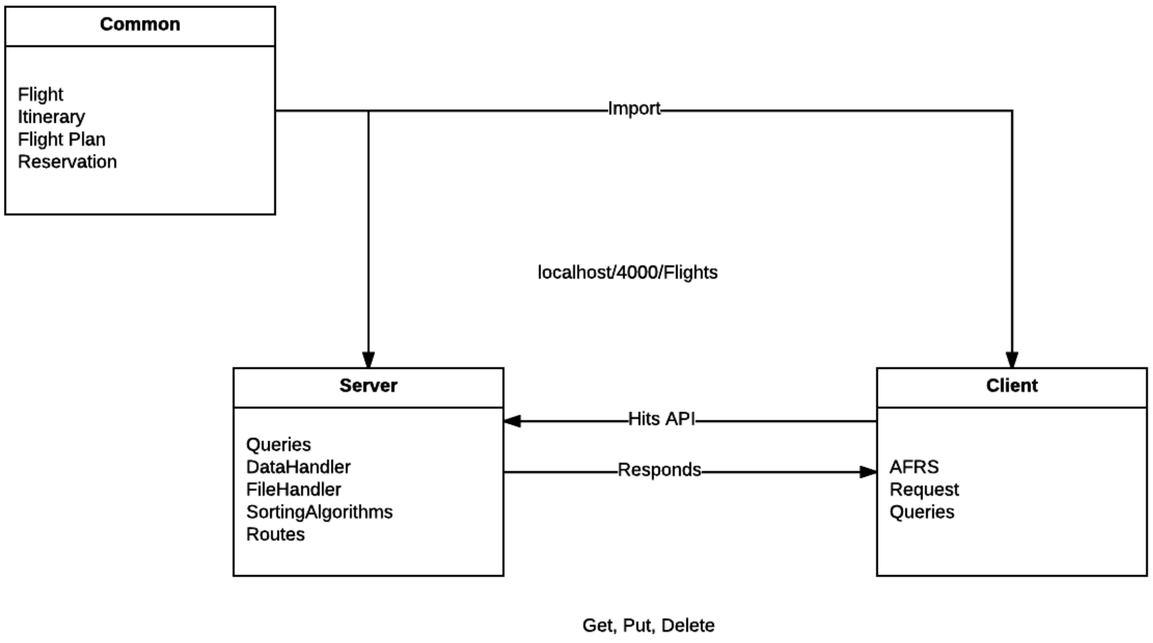
As of the most recent release, the user can no also undo a command recently performed, they can utilize a GUI interface, use data either from the local system or the FAA, as well as being able to have multiple concurrent clients. The system now allows for users across of Treetop Airways to enjoy a more simplistic and useful interface as they manage the reservations throughout the system.

Release 2 Design Updates

Overall the system coming from release 1 was in a good state. The features and requirements from Release 1 were all implemented and functional to the standard. Now when it came to Release 2, there was a need for a break up between client and server interactions. This being that there now needs to be the ability for multiple concurrent clients all performing commands on the system. Because of this, there was a need to break the system out into a client/server/common layout.

The main way to describe this breakout is that the client, cannot directly use any classes from the server, and the server vice versa. However there are parts of the system that both need to access, that should be called the common, which are common classes shared by both systems.

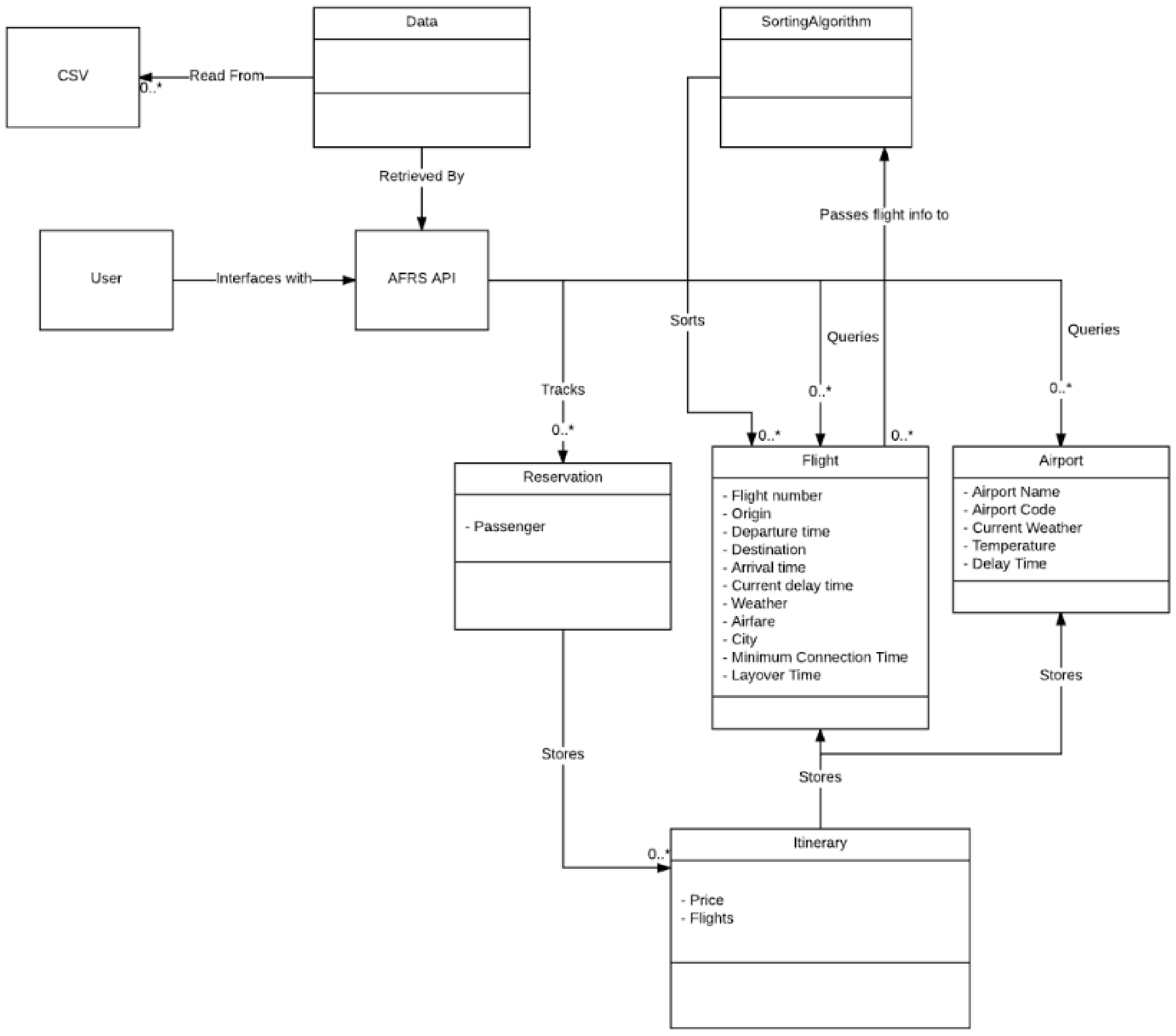
This design required a bit of refactoring and an idea of how we could implement the communication between the clients and the server. The discussion landed on using a web route based web service as the server. This would allow the client to make simple http calls using a RESTful API to a constantly running server.



With this new design in mind, we then had to take into account our current subsystems and how they would fit into this new release, especially with the need to use a series of new design patterns from iterator, proxy and state patterns. State pattern seemed like an easy implementation with the use of FFA or Flat files being a state, and the use of the CMD interface vs the GUI being another. Proxy seemed like another easier implementation given the fact that the API is not called directly by the AFRS client, but instead is called through command pattern style classes, that then, by proxy, call the server. Iterator was a bit harder of a decision to make as to where it should be implemented.

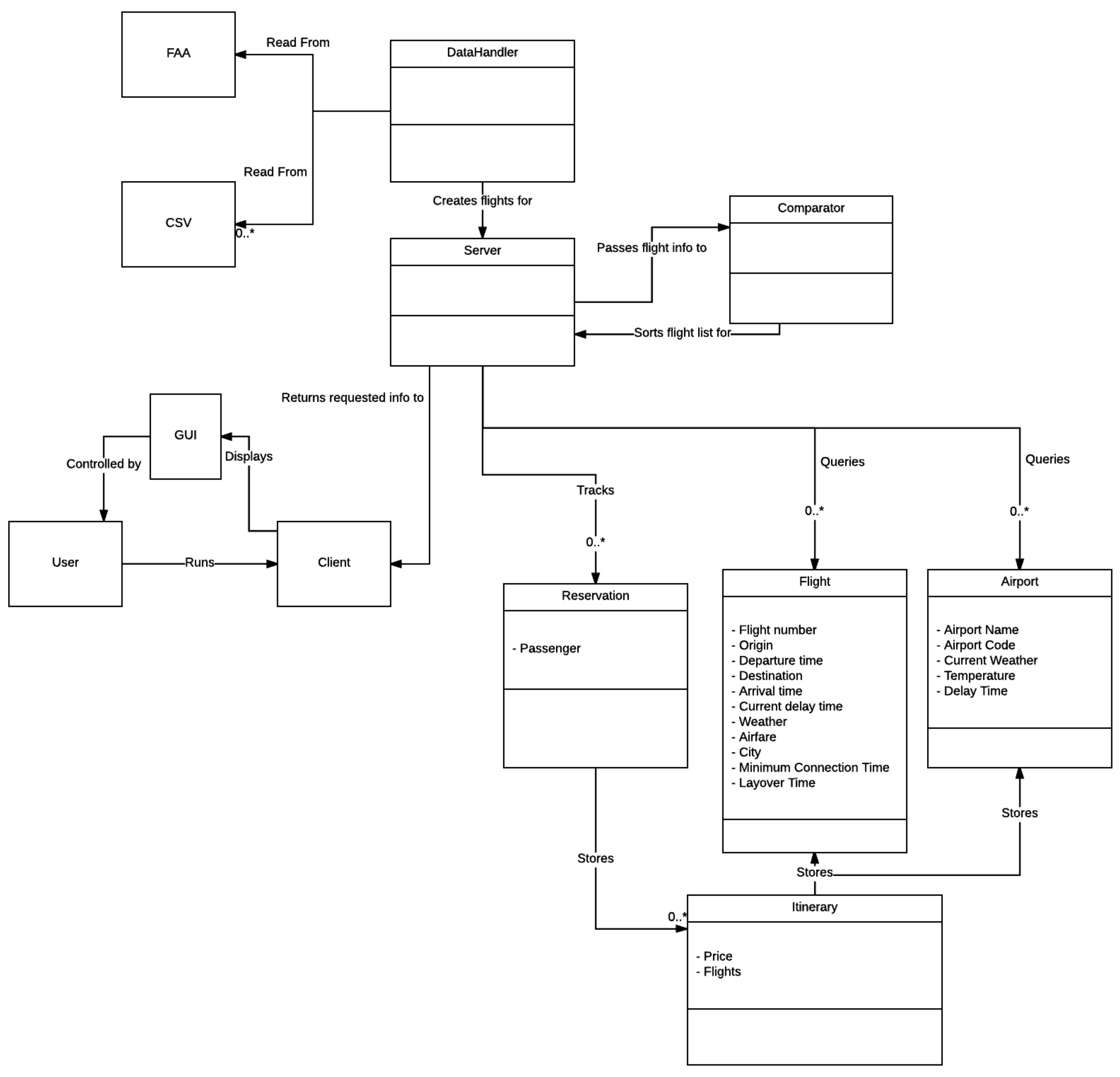
DOMAIN MODEL

Release 1 Release 1 Design was a little underdeveloped. There was a functional system, however it was not very well thought out in the overall design. There was a lot of work to be done on how the system communicated as a whole. In Release 1 the focus was on getting a system that worked, and worked in a logical way instead of on how to design a system that was easy to continue development on after the fact. Because of this the design made sense, but was not very thought out.



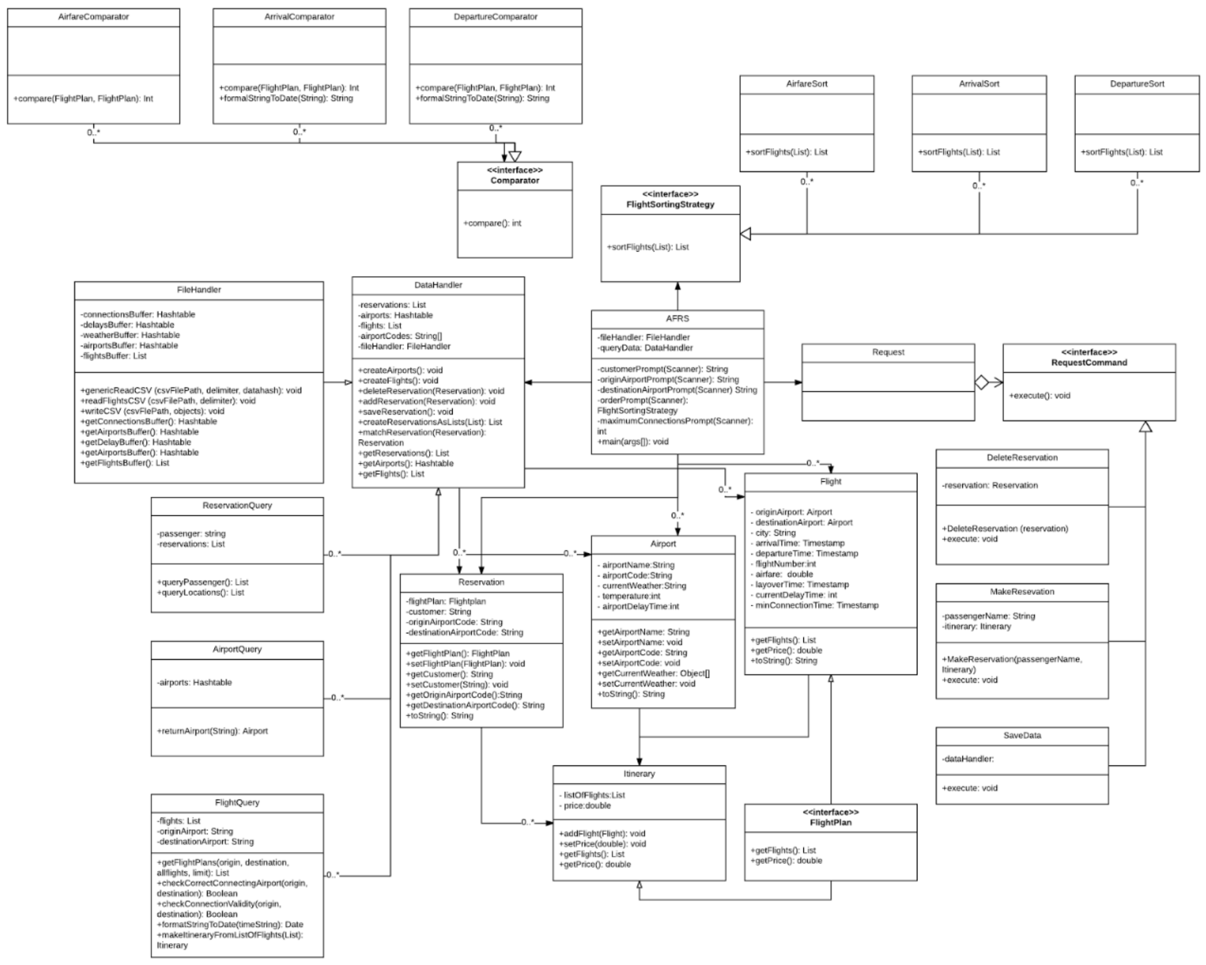
Release 2 Release 2 we took the client/server/common design approach and found a decent amount of success doing so. The design when split into client and server made a lot of sense as to not only how communication works, but also how the system as a whole should be laid

out. By splitting communication we also heavily decoupled the system. The client relies on the server, but the server has no reliance whatsoever on the client, and the client has a lot less information as it only knows what the user and the server give it. Before the client was responsible for all interactions with the datahandler, and the user. This creates a good layer of abstraction as well as decoupling.

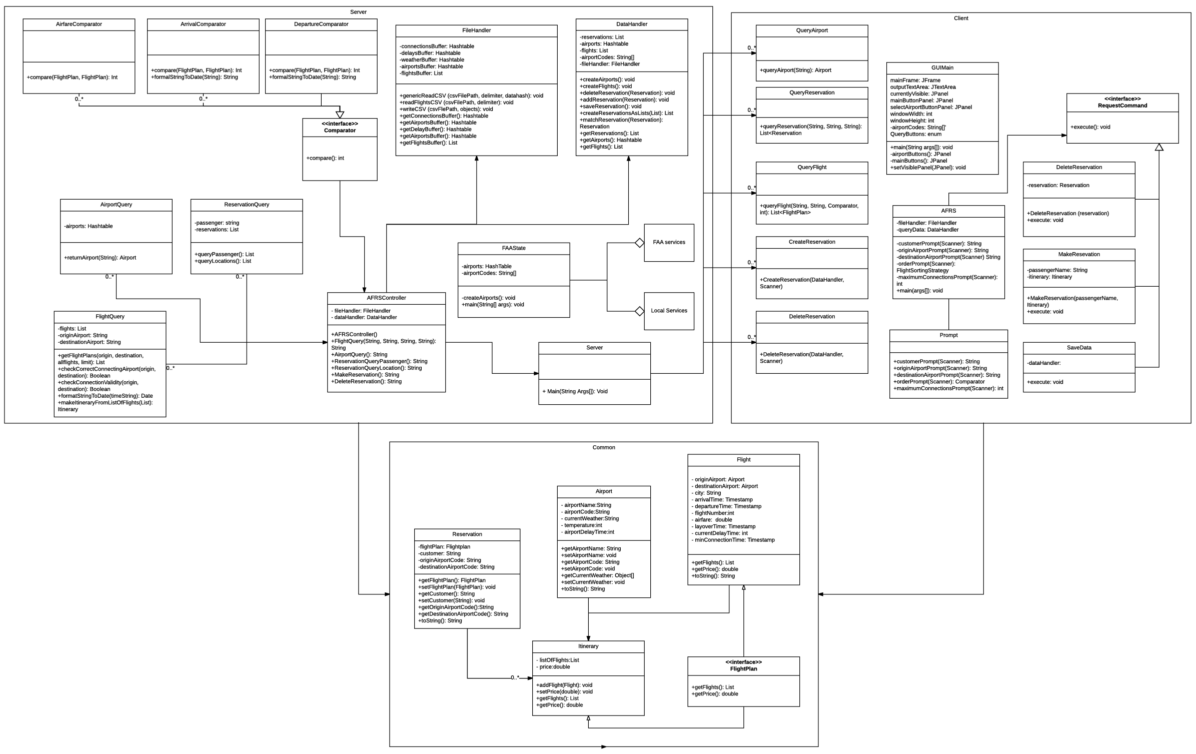


System Architecture

Release 1 At the end of Release 1, we had a large, complex system that was heavily coupled to the AFRS and Datahandler classes. This was made abundantly clear by the fact that when the refactor to the new design was made, the AFRS class was importing a little over half of the classes that had been built in the system.



Release 2 As part of the refactor the design was coupled away from the AFRS class and began the split into the 3 client/server/common systems. This design choice was made in order to decouple the system and allow for multiple concurrent clients with ease. There can be as many clients as needed, as long as the server is running as well. The Server is the most complex of the three major subsystems. A nice benefit of creating this subsystem is decoupling Data Handler and File Handler from some classes that required it in release 1.

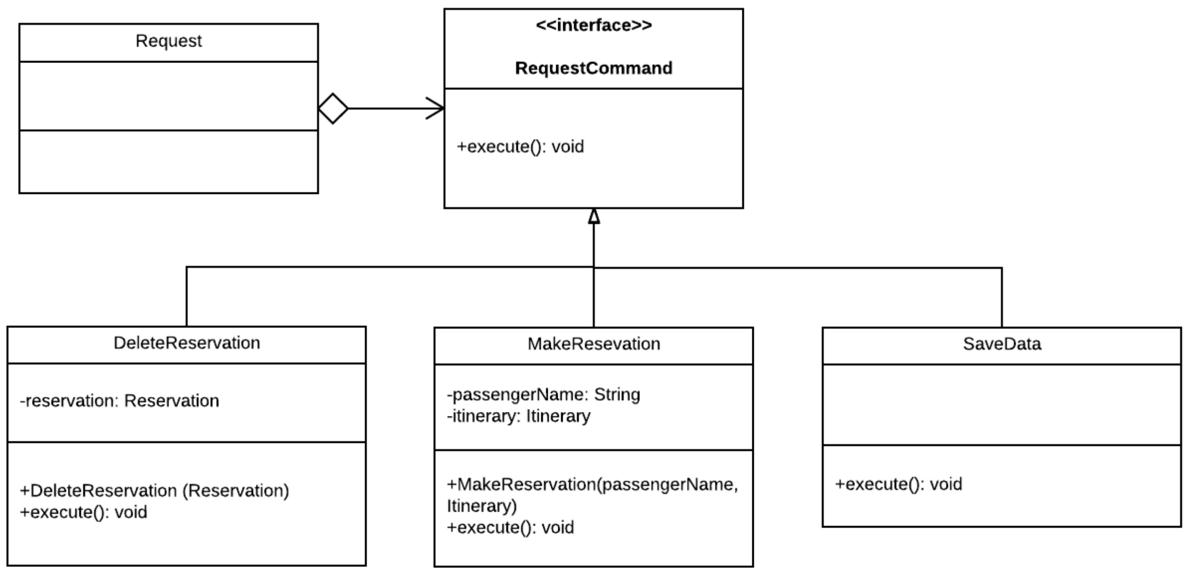

Server
AirfareComparator
+compare(FlightPlan, FlightPlan): Int
ArrivalComparator
+compare(FlightPlan, FlightPlan): Int
+formalStringToDate(String): String
DepartureComparator
+compare(FlightPlan, FlightPlan): Int
+formalStringToDate(String): String
0..*
0..*
<<interface>> Comparator
+compare(): int
FileHandler
-connectionsBuffer: Hashtable
-delaysBuffer: Hashtable
-weatherBuffer: Hashtable
-airportsBuffer: Hashtable
-flightsBuffer: List
+genericReadCSV (csvFilePath, delimiter, datahash): void
+readFlightsCSV (csvFilePath, delimiter): void
+writeCSV (csvFlePath, objects): void
+getConnectionsBuffer(): Hashtable
+getAirportsBuffer(): Hashtable
+getDelayBuffer(): Hashtable
+getAirportsBuffer(): Hashtable
+getFlightsBuffer(): List
DataHandler
-reservations: List
-airports: Hashtable
-flights: List
-airportCodes: String[]
-fileHandler: FileHandler
+createAirports(): void
+createFlights(): void
+deleteReservation(Reservation): void
+addReservation(Reservation): void
+saveReservation(): void
+createReservationsAsLists(List): List
+matchReservation(Reservation): Reservation
+getReservations(): List
+getAirports(): Hashtable
+getFlights(): List
AirportQuery
-airports: Hashtable
+returnAirport(String): Airport
ReservationQuery
-passenger: string
-reservations: List
+queryPassenger(): List
+queryLocations(): List
0..*
0..*
FlightQuery
-flights: List
-originAirport: String
-destinationAirport: String
+getFlightPlans(origin, destination, allflights, limit): List
+checkCorrectConnectingAirport(origin, destination): Boolean
+checkConnectionValidity(origin, destination): Boolean
+formatStringToDate(timeString): Date
+makeItineraryFromListOfFlights(List): Itinerary
0..*
AFRSController
- fileHandler: FileHandler
- dataHandler: DataHandler
+AFRSController()
+FlightQuery(String, String, String, String): String
+AirportQuery(): String
+ReservationQueryPassenger(): String
+ReservationQueryLocation(): String
+MakeReservation(): String
+DeleteReservation(): String
FAAState
-airports: HashTable
-airportCodes: String[]
-createAirports(): void
+main(String[] args): void
FAA services
Local Services
Server
+ Main(String Args[]): Void
Client
QueryAirport
+queryAirport(String): Airport
QueryReservation
+queryReservation(String, String, String): List<Reservation
QueryFlight
+queryFlight(String, String, Comparator, int): List<FlightPlan>
CreateReservation
+CreateReservation(DataHandler, Scanner)
DeleteReservation
+DeleteReservation(DataHandler, Scanner)
0..*
0..*
0..*
0..*
0..*
GUIMain
mainFrame: JFrame
outputTextArea: JTextArea
currentlyVisible: JPanel
mainButtonPanel: JPanel
selectAirportButtonPanel: JPanel
windowWidth: int
windowHeight: int
-airportCodes: String[]'
QueryButtons: enum
+main(String args[]): void
-airportButtons(): JPanel
-mainButtons(): JPanel
+setVisiblePanel(JPanel): void
AFRS
-fileHandler: FileHandler
-queryData: DataHandler
-customerPrompt(Scanner): String
-originAirportPrompt(Scanner): String
-destinationAirportPrompt(Scanner) String
-orderPrompt(Scanner): FlightSortingStrategy
-maximumConnectionsPrompt(Scanner): int
+main(args[]): void
Prompt
+customerPrompt(Scanner): String
+originAirportPrompt(Scanner): String
+destinationAirportPrompt(Scanner): String
+orderPrompt(Scanner): Comparator
+maximumConnectionsPrompt(Scanner): int
<<interface>> RequestCommand
+execute(): void
DeleteReservation
-reservation: Reservation
+DeleteReservation (reservation)
+execute: void
MakeResevation
-passengerName: String
-itinerary: Itinerary
+MakeReservation(passengerName, Itinerary)
+execute: void
SaveData
-dataHandler:
+execute: void
Common
Reservation
-flightPlan: Flightplan
-customer: String
-originAirportCode: String
-destinationAirportCode: String
+getFlightPlan(): FlightPlan
+setFlightPlan(FlightPlan): void
+getCustomer(): String
+setCustomer(String): void
+getOriginAirportCode():String
+getDestinationAirportCode(): String
+toString(): String
Airport
- airportName:String
- airportCode:String
- currentWeather:String
- temperature:int
- airportDelayTime:int
+getAirportName: String
+setAirportName: void
+getAirportCode: String
+setAirportCode: void
+getCurrentWeather: Object[]
+setCurrentWeather: void
+toString(): String
Flight
- originAirport: Airport
- destinationAirport: Airport
- city: String
- arrivalTime: Timestamp
- departureTime: Timestamp
- flightNumber:int
- airfare: double
- layoverTime: Timestamp
- currentDelayTime: int
- minConnectionTime: Timestamp
+getFlights(): List
+getPrice(): double
+toString(): String
Itinerary
- listOfFlights:List
- price:double
+addFlight(Flight): void
+setPrice(double): void
+getFlights(): List
+getPrice(): double
0..*
<<interface>> FlightPlan
+getFlights(): List
+getPrice(): double

SUBSYSTEMS

Client Requests

Class Diagram

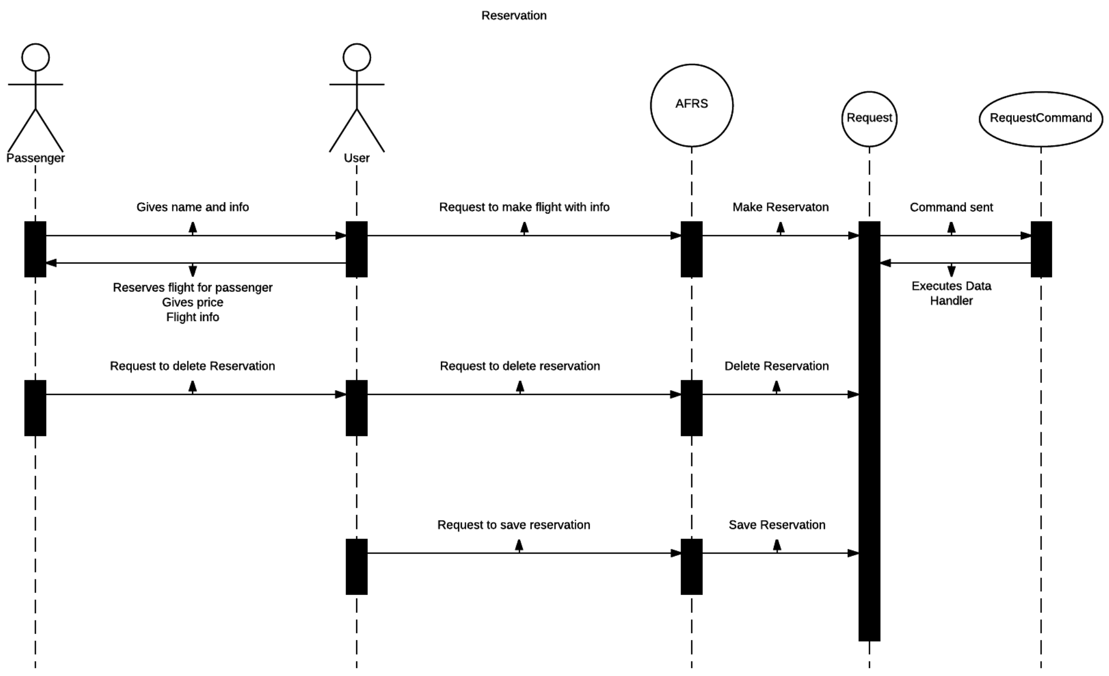
Release 1 In release 1, the client had 3 actions that could make through the request classes. They could make, delete or save reservations. The requirements only required that the user was able to make and delete a reservation, and that it would persist throughout the system. In order to make up for the fact that undo was not present in the system, saving the reservations was a separate command to allow for not saving changes that were not wanted. This was also a pretty poor implementation of the command pattern, as the invoker was never used, and each command was instantiated and removed at use.



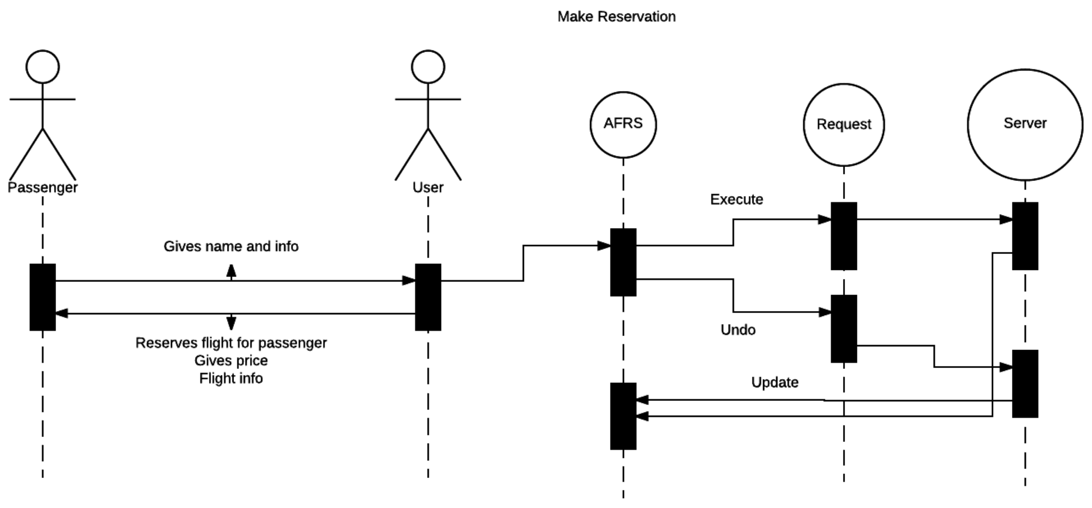
Release 2 During release 2, the command pattern is moved around and used in a few places. The requests from Release 1, are still in the client, and provide the undo/redo functionality as they are now meant to by the requirements. However now they no longer interact with the datahandler directly, they now call routes in the server to perform commands. Completely invisible to the user, or the client, the actions are then performed on the server. Now as well as providing a command pattern, this section of the design also provides a proxy pattern for giving the client a place to hit, before all actions are done elsewhere. A design choice made was to use the command and proxy patterns heavily modified here in order to make it work as we designed.

Sequence Diagrams

Release 1



Release 2 In Release 2 we omitted the Invoker entirely as it was unnecessary. We also had to now account for undoing of commands. With this in mind, it was actually pretty easy to implement simple undo/redo functionality as long as we held a request object in memory to allow us to call the undo/execute commands.



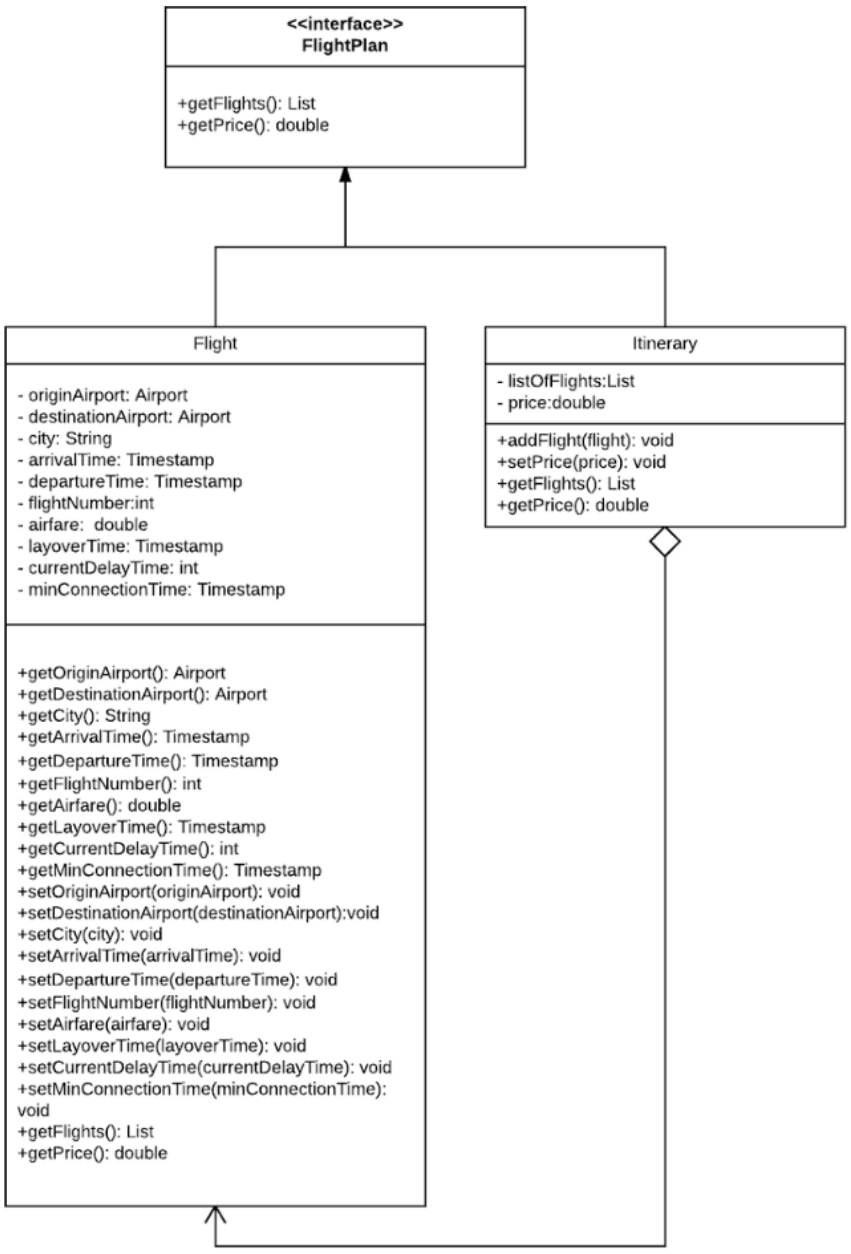
GoF Card

Name: Request		GoF pattern: Command
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
RequestCommand	Command	The overall command which could be used to reuse an existing request as another one. Although not currently used in this fashion, it would be fairly easy to make the change.
MakeReservation	Concrete Command	Makes a new reservation object and adds it to the system. This does so without returning anything to the user, or informing the user about what commands are being performed.
DeleteReservation	Concrete Command	Deletes reservation object from the system. This does so by calling the class in the datahandler without exposing that to the invoker.
Deviations from the Pattern: • No reflection/pointers involved for target method. • Commands have more distinct methods. • No Invoker class.		
Requirements being covered: • The AFRS shall track reservations. • The system shall allow a client to make a reservation for an itinerary contained in the most recent query for flight information. • The system shall allow a client to query for reservations for a passenger. • The system shall allow a client to delete a reservation for a passenger.		

Itineraries

Class Diagram

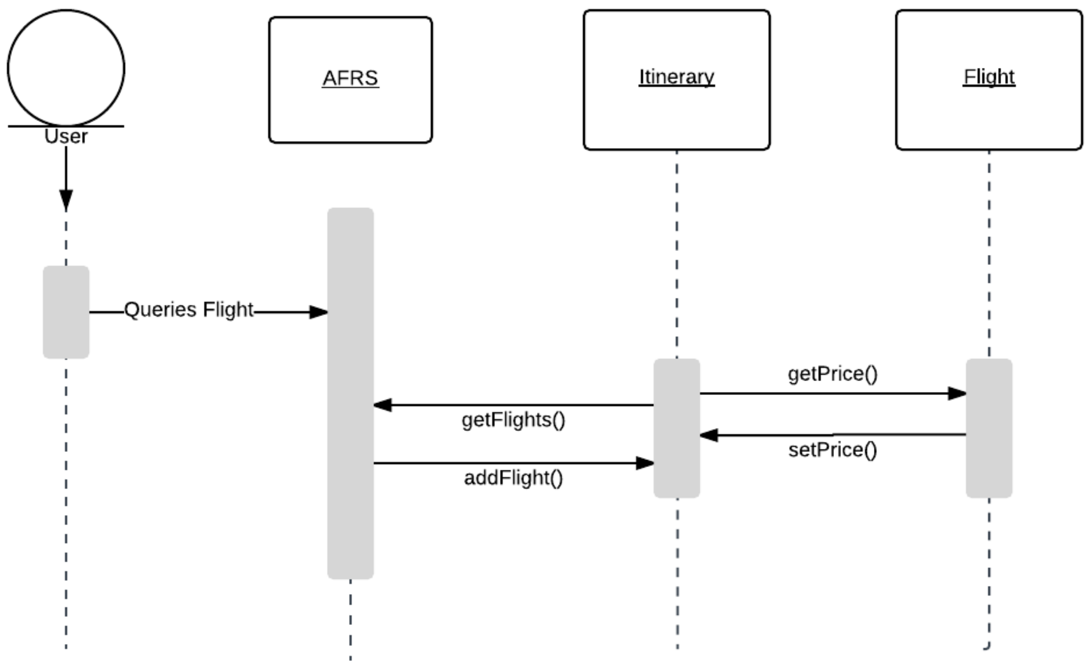
Release 1 & 2 The design for Flights and Itineraries is one of the subsystems that is the most easily understood. Flights and Itineraries share a common interface, and therefore can be used interchangeably by the rest of the system. This use of the composite pattern is incredibly useful here, and because of that, no changes were made between R1 and R2.



<<interface>>
FlightPlan
+getFlights(): List
+getPrice(): double
Flight
- originAirport: Airport
- destinationAirport: Airport
- city: String
- arrivalTime: Timestamp
- departureTime: Timestamp
- flightNumber:int
- airfare: double
- layoverTime: Timestamp
- currentDelayTime: int
- minConnectionTime: Timestamp
+getOriginAirport(): Airport
+getDestinationAirport(): Airport
+getCity(): String
+getArrivalTime(): Timestamp
+getDepartureTime(): Timestamp
+getFlightNumber(): int
+getAirfare(): double
+getLayoverTime(): Timestamp
+getCurrentDelayTime(): int
+getMinConnectionTime(): Timestamp
+setOriginAirport(originAirport): void
+setDestinationAirport(destinationAirport):void
+setCity(city): void
+setArrivalTime(arrivalTime): void
+setDepartureTime(departureTime): void
+setFlightNumber(flightNumber): void
+setAirfare(airfare): void
+setLayoverTime(layoverTime): void
+setCurrentDelayTime(currentDelayTime): void
+setMinConnectionTime(minConnectionTime): void
+getFlights(): List
+getPrice(): double
Itinerary
- listOfFlights:List
- price:double
+addFlight(flight): void
+setPrice(price): void
+getFlights(): List
+getPrice(): double

Sequence Diagrams

Release 1 & 2 In Release 2, we added server route functionality to this subsystem in order to fit it in with the multi-concurrent user system.



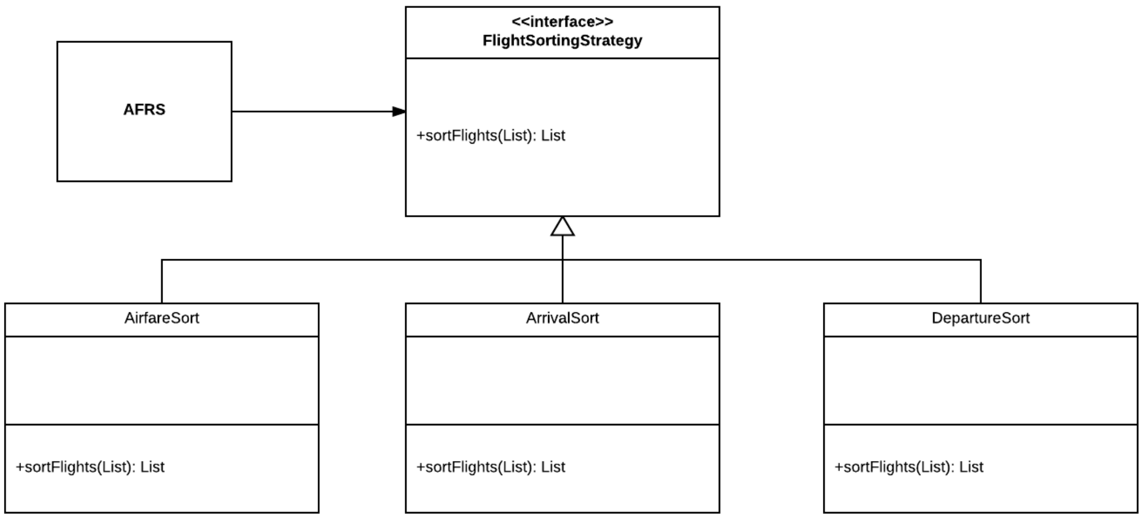
GoF Card

Name: Itineraries		GoF pattern: Composite
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
Flight Plan	Component	The component interface to uniform the leaf and composite.
Flight	Leaf	The actual flight object as it exists in the database. This will include all information needed to understand a flight.
Itinerary	Composite	The itinerary object, with a list of flights, and the total price for each of the flights inside of it. This itinerary is a possible set of flights to get from one destination to another.
Deviations from the Pattern: • Leaf has many more encapsulated attributes than composite element. • Composite does not contain Composites. It will only contain leafs.		
Requirements being covered: • The system shall store flight data for all flights between the cities in TTA's route network. Flight data shall consist of: origin airport, destination airport, departure time, arrival time, flight number, and airfare. An itinerary is a list of Flights that also stores the total airfare of each flight. This is used when a client creates a reservation with multiple legs.		

Sorting Algorithms

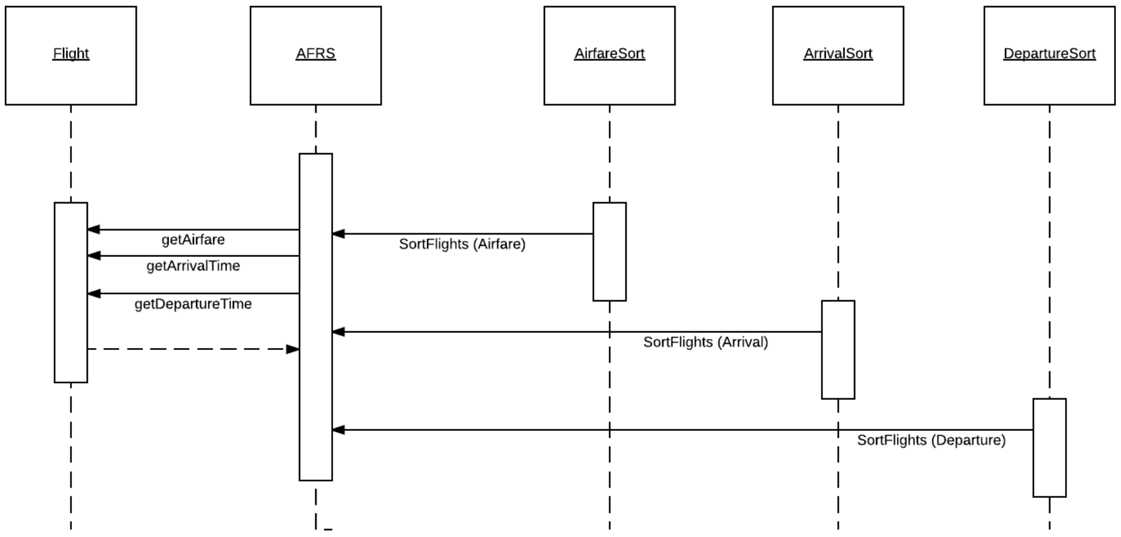
Class Diagram

Release 1 & 2 In Release 1, there were a series of classes whose sole purpose was to return a comparator. During the discussion over the R1 design, it was determined that those extra classes were unnecessary to the use of the strategy pattern. This was creating a series of classes that served no extra functionality, and simply abstracted a part of the system that never needed to be abstracted.



Sequence Diagrams

Release 1 & 2



GoF Card

Name: Sorting Algorithms		GoF pattern: Strategy
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
AFRS	Context	Executes the strategy
FlightSortingStrategy	Strategy	An interface for multiple strategies for sorting flights
AirfareSort	ConcreteStrategyA	An algorithm for sorting airfare. It orders all flights and itineraries by using a Comparator implementation that compares price of two flights. Flights are ordered from least to most expensive.
ArrivalSort	ConcreteStrategyB	An algorithm for sorting arriving times. It orders FlightPlans by using a Comparator implementation that compares the Date objects of the last flight in the FlightPlans arrival time.
DepartureSort	ConcreteStrategyC	An algorithm for sorting departing times. Using an implementation of the Comparator interface that compares two Date objects for the departure of the first flight in a FlightPlan.

Deviations from the Pattern:

- Each algorithm essentially does the same work with a different comparator rather than a completely different algorithm. However, the client still has the power to change a backend algorithm and it is done on the fly.

Requirements being covered:

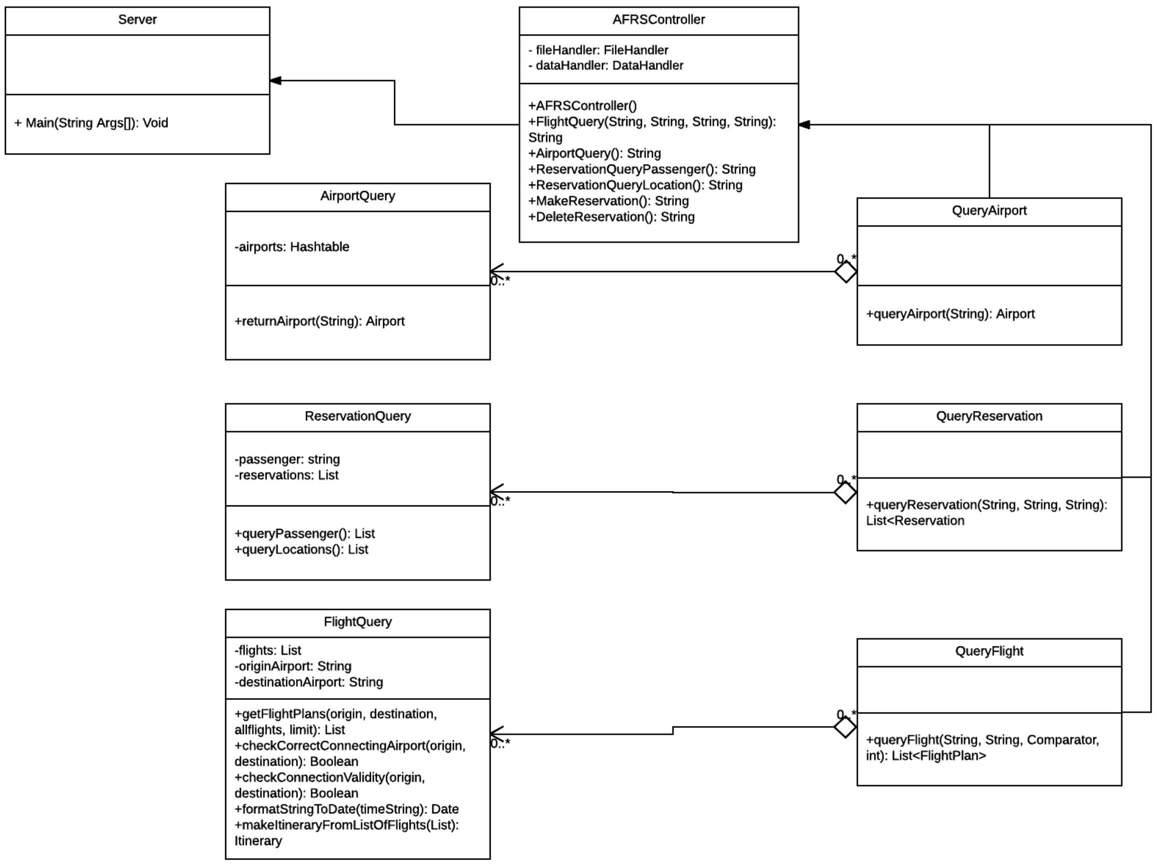
- Sorting of Flights and Itineraries given an option to sort them.

Proxy

In the transition to a client-server subsystem, we implemented a controller for the user to request information. With the proxy pattern, the system gives the client methods they can call, then translates that call to server. This way, the client doesnt need to know where the data is or how to explicitly fetch it.

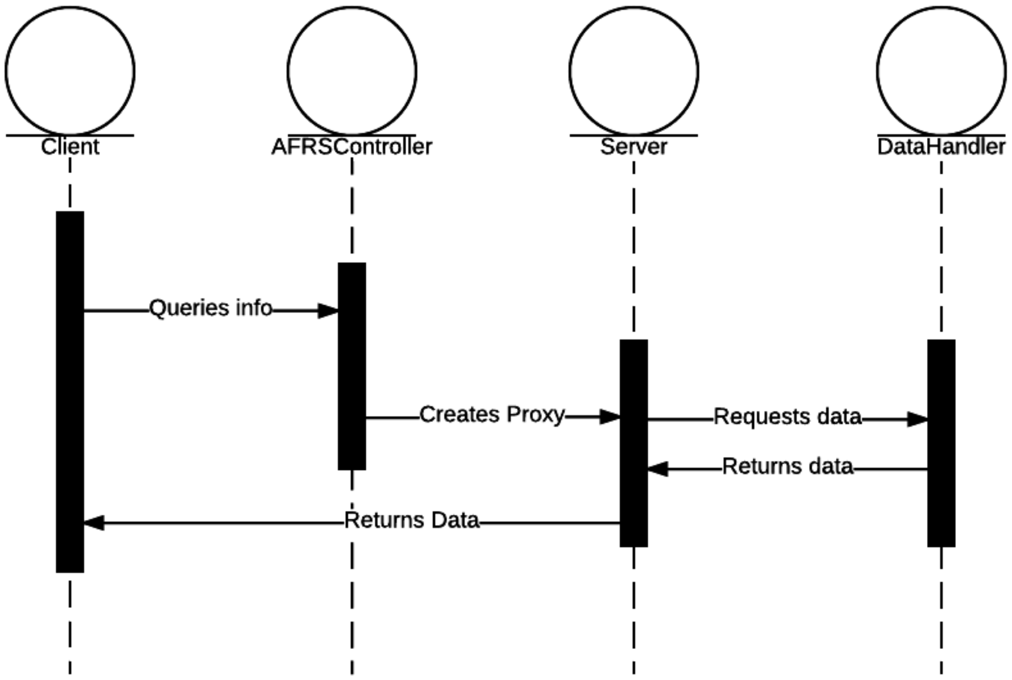
Class Diagram

Release 2


Server
+ Main(String Args[]): Void
AFRSController
- fileHandler: FileHandler
- dataHandler: DataHandler
+AFRSController()
+FlightQuery(String, String, String, String): String
+AirportQuery(): String
+ReservationQueryPassenger(): String
+ReservationQueryLocation(): String
+MakeReservation(): String
+DeleteReservation(): String
AirportQuery
-airports: Hashtable
+returnAirport(String): Airport
QueryAirport
+queryAirport(String): Airport
ReservationQuery
-passenger: string
-reservations: List
+queryPassenger(): List
+queryLocations(): List
QueryReservation
+queryReservation(String, String, String): List<Reservation
FlightQuery
-flights: List
-originAirport: String
-destinationAirport: String
+getFlightPlans(origin, destination, allflights, limit): List
+checkCorrectConnectingAirport(origin, destination): Boolean
+checkConnectionValidity(origin, destination): Boolean
+formatStringToDate(timeString): Date
+makeItineraryFromListOfFlights(List): Itinerary
QueryFlight
+queryFlight(String, String, Comparator, int): List<FlightPlan>
0..*
0..*
0..*
0..*
0..*
0..*

Sequence Diagrams

Release 2



GoF Card

Name: Proxy		GoF pattern: Proxy
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
CreateReservation-Proxy	Proxy	Makes a call to the Make Reservation Route on the server to create a reservation with the matching object information.
/MakeReservation-Query	Real Subject	Route for performing the actual reservation creation. This is done on the server regardless of which client made the request. Once it has made the request, all clients will be able to see this reservation in the system.
DeleteReservation-Proxy	Proxy	Makes a call to the Delete Reservation Route on the server to delete the reservation matching the object information sent over.
/DeleteReservation-Query	Real Subject	Route for performing the actual reservation deletion. This is done on the server regardless of which client made the request. Once it has deleted the reservation all clients will no longer be able to see the reservation in the system.
QueryAirport	Proxy	Makes a call to the Airport Query on the Server and returns the airport object that the server returns

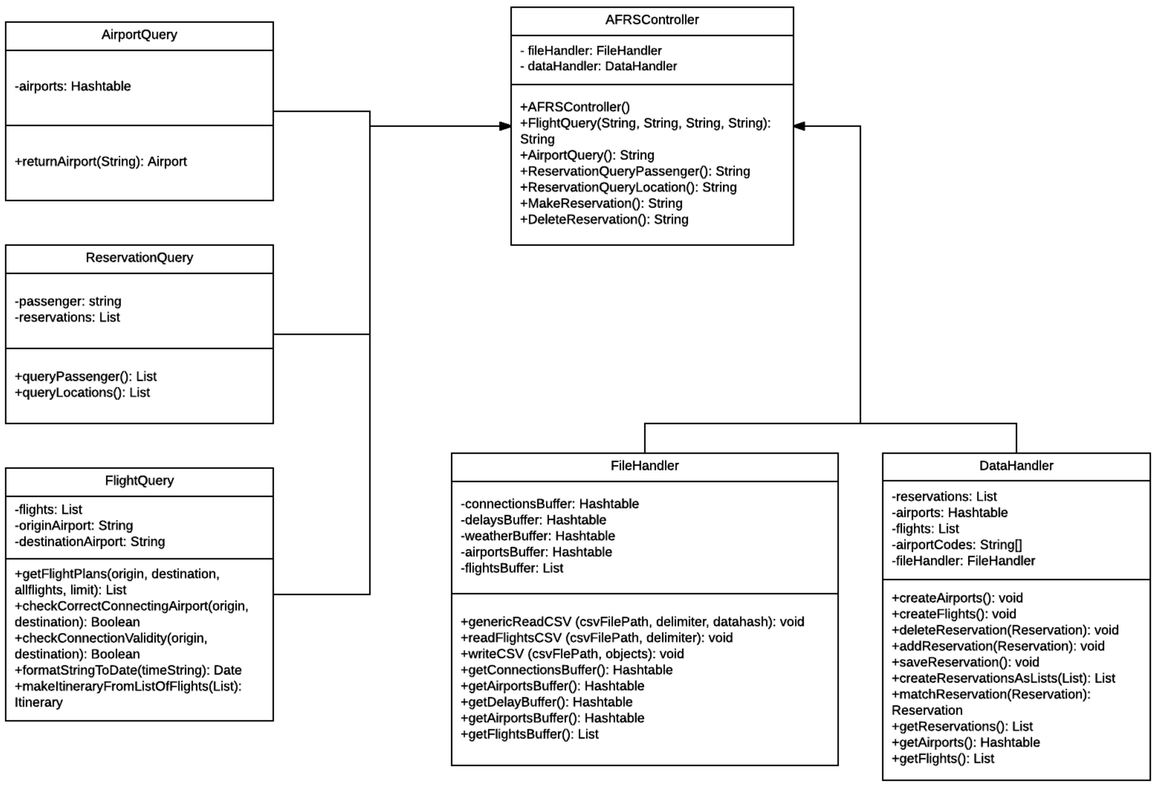
Class	Role in GoF pattern	Participant's contribution in the context of the application
/AirportQuery	Real Subject	Searches either the FAA or local files to return the airport object of the request, then returns it.
QueryFlight	Proxy	Makes a call to the Flight Query on the server and receives the flights and itineraries that exist on the server given the information provided.
/FlightQuery	Real Subject	Searches through its collection of flights and itineraries to return a list of Flight-Plans to the client making the request.
QueryReservation	Proxy	Makes a call to the Query Reservation on the server and recieves Reservations given a passenger name.
/ReservationQueryPas-senger	Real Subject	Searches through existing reservations for any reservations matching the queried passenger name and returns all matching objects to the client.
Deviations from the Pattern: • Proxy and subject classes do not implement an interface.		
Requirements being covered: • The AFRS supports multiple clients but not truly concurrent clients.		

RESTFul Web Service

One significant issue coming from release 1 was the coupling of our queries to the data handler. With the introduction of a new multi-user requirement, we implemented a Facade Pattern, having our server controller (AFRSController) serve as the facade for all server subsystems to interact in one place. What keeps it from being anti-pattern is how it is coupled; none of the subsystems need to know about AFRSController, while direct coupling between queries and datahandler is kept minimal.

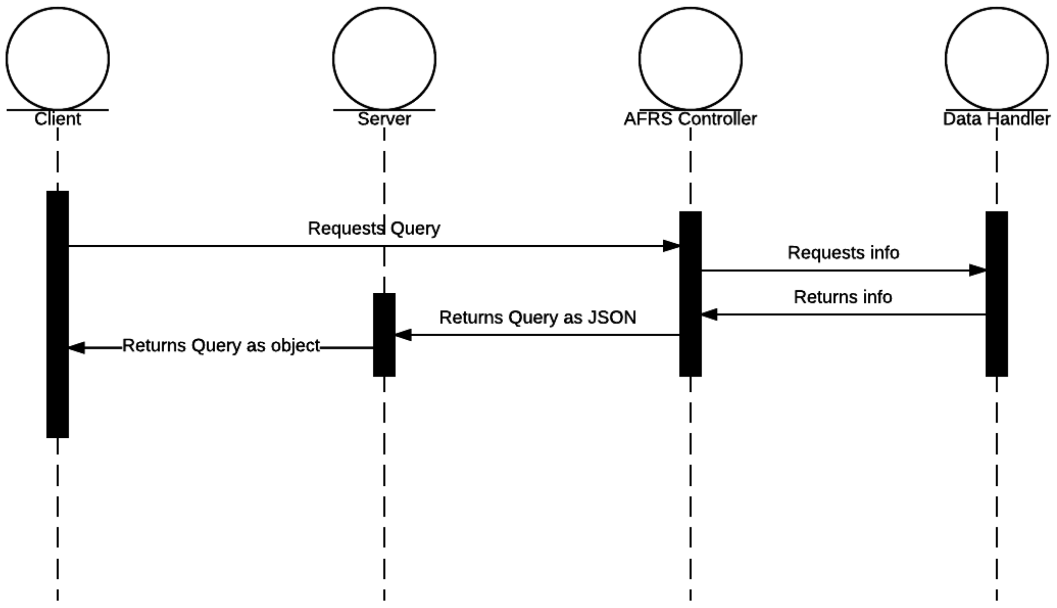
Class Diagram

Release 2


AirportQuery
-airports: Hashtable
+returnAirport(String): Airport
ReservationQuery
-passenger: string
-reservations: List
+queryPassenger(): List
+queryLocations(): List
FlightQuery
-flights: List
-originAirport: String
-destinationAirport: String
+getFlightPlans(origin, destination, allflights, limit): List
+checkCorrectConnectingAirport(origin, destination): Boolean
+checkConnectionValidity(origin, destination): Boolean
+formatStringToDate(timeString): Date
+makeItineraryFromListOfFlights(List): Itinerary
AFRSController
- fileHandler: FileHandler
- dataHandler: DataHandler
+AFRSController()
+FlightQuery(String, String, String, String): String
+AirportQuery(): String
+ReservationQueryPassenger(): String
+ReservationQueryLocation(): String
+MakeReservation(): String
+DeleteReservation(): String
FileHandler
-connectionsBuffer: Hashtable
-delaysBuffer: Hashtable
-weatherBuffer: Hashtable
-airportsBuffer: Hashtable
-flightsBuffer: List
+genericReadCSV (csvFilePath, delimiter, datahash): void
+readFlightsCSV (csvFilePath, delimiter): void
+writeCSV (csvFlePath, objects): void
+getConnectionsBuffer(): Hashtable
+getAirportsBuffer(): Hashtable
+getDelayBuffer(): Hashtable
+getAirportsBuffer(): Hashtable
+getFlightsBuffer(): List
DataHandler
-reservations: List
-airports: Hashtable
-flights: List
-airportCodes: String[]
-fileHandler: FileHandler
+createAirports(): void
+createFlights(): void
+deleteReservation(Reservation): void
+addReservation(Reservation): void
+saveReservation(): void
+createReservationsAsLists(List): List
+matchReservation(Reservation): Reservation
+getReservations(): List
+getAirports(): Hashtable
+getFlights(): List

Sequence Diagrams

Release 2



GoF Card

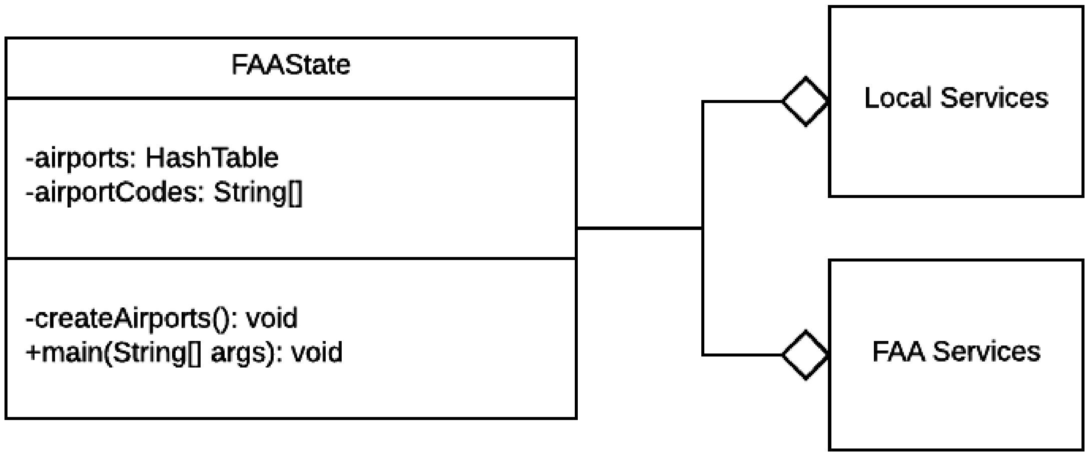
Name: RESTFul Web Service		GoF pattern: Facade
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
AFRSController	Facade	Encapsulates all queries with datahandler and calls queries server-side using the datahandler.
AirportQuery	SubsystemA	Returns an airport object
ReservationQuery	SubsystemA	Returns a list of reservations by passenger name or origin and destination airports
FlightQuery	SubsystemA	Returns a list of flight plans
DataHandler	SubsystemB	Provides queried data from local or FAA
Deviations from the Pattern: • SubsystemBs DataHandler is instantiated in the SubsystemA classes directly for construction. • Facade does add some new functionality related to server interaction • Facade is not an interface		
Requirements being covered: • Server and client interactions enabling multi-concurrent users.		

FAA or Local State

One new requirement in R2 was to allow the user to fetch information from FAA services instead of local. The system initially loads from local anyway, which we believed made it a good fit for the State Pattern. It has an obvious default state (local) and can transition to FAA on request and back. Additionally it only loads local by default, which makes the system load faster than having to load both simultaneously (in case the user never wishes to fetch from FAA). This also makes state transitions after the first one faster since local and FAA remain loaded.

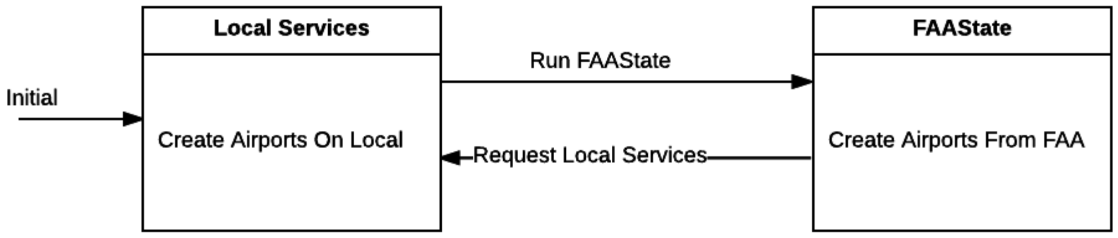
Class Diagram

Release 2



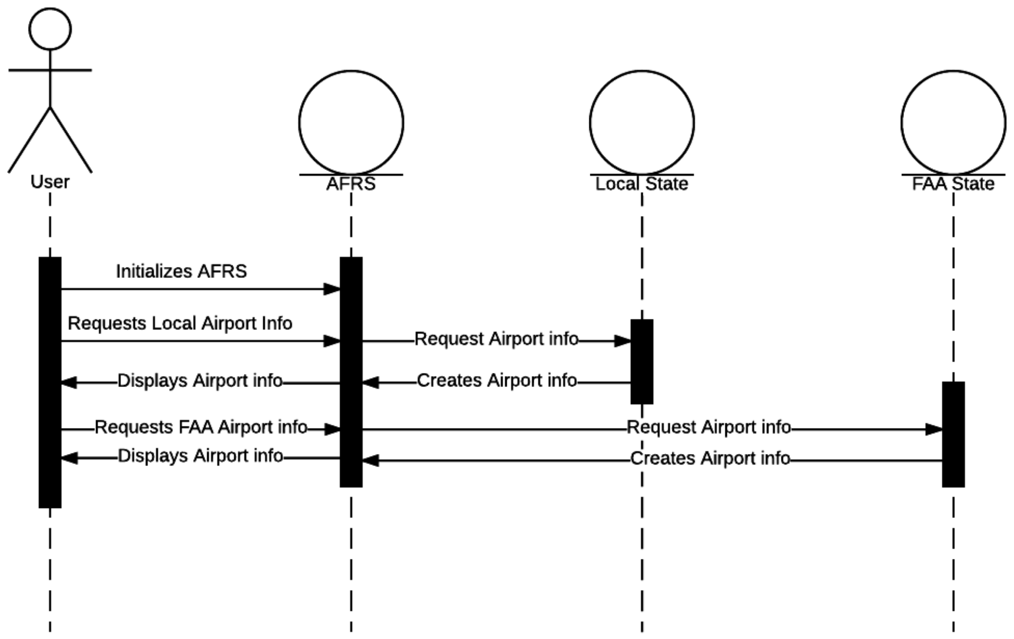
State Diagram

Release 2



Sequence Diagrams

Release 2



GoF Card

Name: FAA or Local State		GoF pattern: State
Participants		
Class	Role in GoF pattern	Participant's contribution in the context of the application
FAAState	ConcreteState	Enables system to read data from FAA service only if the client requests it.
Deviations from the Pattern: • State does not implement an interface		
Requirements being covered: • Provides the user the ability to view airport information either from local services or FAA.		

Status of the Implementation

Overall the implementation is about minimizing places for user error. Out system decided against doing single string requests and instead went for a series of prompts to the user to create a request that cannot be made incomplete. Partial requests are eliminated. There is no need for semicolons (;). With these things in mind most of the logic can exist on the backend instead of up front on the user interface.

User Interface Implementation

- Text interface provides users a sequence of preset options. The reasons we implemented it this way are:
  - This eliminates the need for the user to memorize commands.
  - Request process is sped up.
  - This reduces the possibility for input error (spelling, format, accidental termination).
  - Less testing is needed.
- GUI interface is available to alleviate the fallbacks of using a terminal based interface.
  - User input is handled by more mouse clicks than keyboard input
  - Errors are reduced by prompts being provided with clearer displays

CSV Implementation

- A small subsystem was created to handle data read/writes. This system stores data in hash tables as an alternative to database:
  - A FileHandler class reads from CSVs and writes to other CSVs.
  - A DataHandler class holds methods to manipulate data from FileHandler (i.e Flights, Reservations)

Client/Server/Common

- After the release of Release 1, the system was split into client, server and common systems. This design change meaning that no classes inside client were used by server, and vice versa, with the only classes being shared being from common.

  - This decouples client and server.

RESTful Web Service

- Our choice for handling multiple concurrent user support was implementing Spring RESTful Web Service via Gradle.
  - This simplified creating routes heavily. Every command has a small route implementation to return a parsable JSON representation.
  - We decided this outweighed the difficulty of integrating Gradle into our project.

APPENDIX

Common

Class: Airport	
Responsibilities: • Object to store all information about each Airport.	
Collaborators:	
Users:	Used by: AirportQuery, DataHandler, AFRS, AFRSController
Author: Devin Matte	
Package: Models	

Class: Flight	
Responsibilities: • Object to store all information about any given flight to use in building itineraries and reservations	
Collaborators:	
Users: FlightPlan	Used by: Itinerary, AFRSController
Author: Amanda Ramos	
Package: Models	

Class: Itinerary	
Responsibilities: • Collect a series of flights to make a full flight including connections from one airport to another.	
Collaborators:	
Users: Flight, FlightPlan	Used by: DataHandler, FlightQuery, AFRSController
Author: Nicholas Montemorano, Devin Matte	
Package: Models	

Class: Reservation	
Responsibilities: • A flight, or itinerary of flights for a customer to fly. Has the customer, origin, destination and corresponding flight plan	
Collaborators:	
Users: Flight, FlightPlan	Used by: AFRS, DataHandler, AFRSController
Author: Nicholas Montemorano, Oren Rosin	
Package: Models	

Server

Class: AFRSController	
Responsibilities: • Creates routes for queries made by the client. Queries include Flight plans, Airports, and Reservations as well as executing methods for making and deleting reservations.	
Collaborators:	
Users: FlightQuery, AirportQuery, ReservationQuery, DataHandler, FileHandler, Flight, FlightPlan, Itinerary, Resevation, AirfareComparator, ArrivalComparator, DepartureComparator	Used by: Server
Author: Oren Rosin, Devin Matte	
Package: RESTfulAPI	

Class: Server	
Responsibilities: • Boots Spring Web Service, allowing the routes to be created.	
Collaborators:	
Users: AFRSController	Used by: Server
Author: Oren Rosin, Devin Matte	
Package: RESTfulAPI	

Class: FileHandler	
Responsibilities: • Read/Write for CSV files. Stores data in hash tables.	
Collaborators:	
Users: None	Used by: AFRSController
Author: Nicholas Montemorano	
Package: DataHandler	

Class: DataHandler	
Responsibilities: • Holds methods to manipulate data in FileHandler	
Collaborators:	
Users: Airport, Flight, Itinerary, Reservation	Used by: AFRS, MakeReservation, DeleteReservation
Author: Nicholas Montemorano	
Package: DataHandler	

Class: AirfareComparator	
Responsibilities: • Implements the Comparator interface, meaning it must implement the compare method that returns an integer. It takes two FlightPlan objects, retrieves their airfare and returns a positive or negative integer to indicate which airfare is higher.	
Collaborators:	
Users: FlightPlan	Used by: None
Author: Nicholas Montemorano	
Package: Comparators	

Class: DepartureComparator	
Responsibilities: • Implements the Comparator interface by creating a compare method that takes two FlightPlan objects. The FlightPlans contain a departure time as a String. This class converts each date String to a Date object and compares the two, returning a positive or negative integer.	
Collaborators:	
Users: FlightPlan	Used by: None
Author: Nicholas Montemorano	
Package: Comparators	

Class: ArrivalComparator	
Responsibilities: • Implements the Comparator interface by creating a compare method that takes two FlightPlan objects. The FlightPlans contain an arrival time of each flight as a String. This class converts the last flights date from a string to a Date object and compares the two, returning a positive or negative integer.	
Collaborators:	
Users: FlightPlan	Used by: None
Author: Nicholas Montemorano	
Package: Comparators	

Class: AirportQueryServer	
Responsibilities: • Creates and returns an airport object with relevant information to that airport, including name, code, weather, temperature and delay time	
Collaborators:	
Users: Datahandler, Airport	Used by: AFRSController
Author: Devin Matte	
Package: Queries	

Class: FlightQueryServer	
Responsibilities: • Creates and returns a list of flight plans, containing information for flight number, origin, destination, and price of each flight.	
Collaborators:	
Users: DataHandler, Flight, FlightPlan, Itinerary, DepartureComparator	Used by: AFRSController
Author: Devin Matte	
Package: Queries	

Class: ReservationQueryServer	
Responsibilities: • Creates and returns a list of reservation objects based on either a) the requested passenger name or b) the requested origin and destinations of the reservation.	
Collaborators:	
Users: DataHandler, Reservation	Used by: AFRSController
Author: Devin Matte	
Package: Queries	

Client

Class: AFRS	
Responsibilities: • AFRS is the main class. This is what is run when the user runs the program. It prompts for all user input and does most of the input handling.	
Collaborators:	
Users: CreateReservationProxy, DeleteReservationProxy QueryAirport, QueryFlight, QueryReservation, Airport, FlightPlan, Reservation, Prompt	Used by: None
Author: Devin Matte, Amanda Ramos	

Class: MakeReservation	
Responsibilities: • A command which tells the API to create a reservation given one passed to the command.	
Collaborators:	
Users: Reservation	Used by: CreateReservationProxy
Author: Devin Matte	
Package: Request	

Class: DeleteReservation	
Responsibilities: • Tells the API to delete a reservation from the system memory. It deletes the one passed in the constructor.	
Collaborators:	
Users: Reservation	Used by: DeleteReservationProxy
Author: Devin Matte	
Package: Request	

Class: QueryAirport	
Responsibilities: • Given a airport code, return the airport object that corresponds to it. It simply returns the result.	
Collaborators:	
Users: Airport	Used by: AFRS, DataHandler
Author: Devin Matte	
Package: Proxy	

Class: QueryFlight	
Responsibilities: • Queries for and returns flights given a set of data. Allows for the data to be ordered or set a maximum number of connections. This is done through the strategy pattern.	
Collaborators:	
Users: Flight, FlightPlan, Itinerary	Used by: Flight, Itinerary, DepartureSort, DataHandler
Author: Devin Matte	
Package: Proxy	

Class: QueryReservation	
Responsibilities: • Makes a query given nothing but the passenger and returns a list of all reservations for that passenger	
Collaborators:	
Users: Flight, FlightPlan, Itinerary, Reservation	Used by: None
Author: Devin Matte	
Package: Proxy	

Class: Prompt	
Responsibilities: • Lists the available options to the user via Scanner	
Collaborators:	
Users: Scanner, AFRS, AirfareComparator, ArrivalComparator, DepartureComparator	Used by: None
Author: Devin Matte	