AFRS

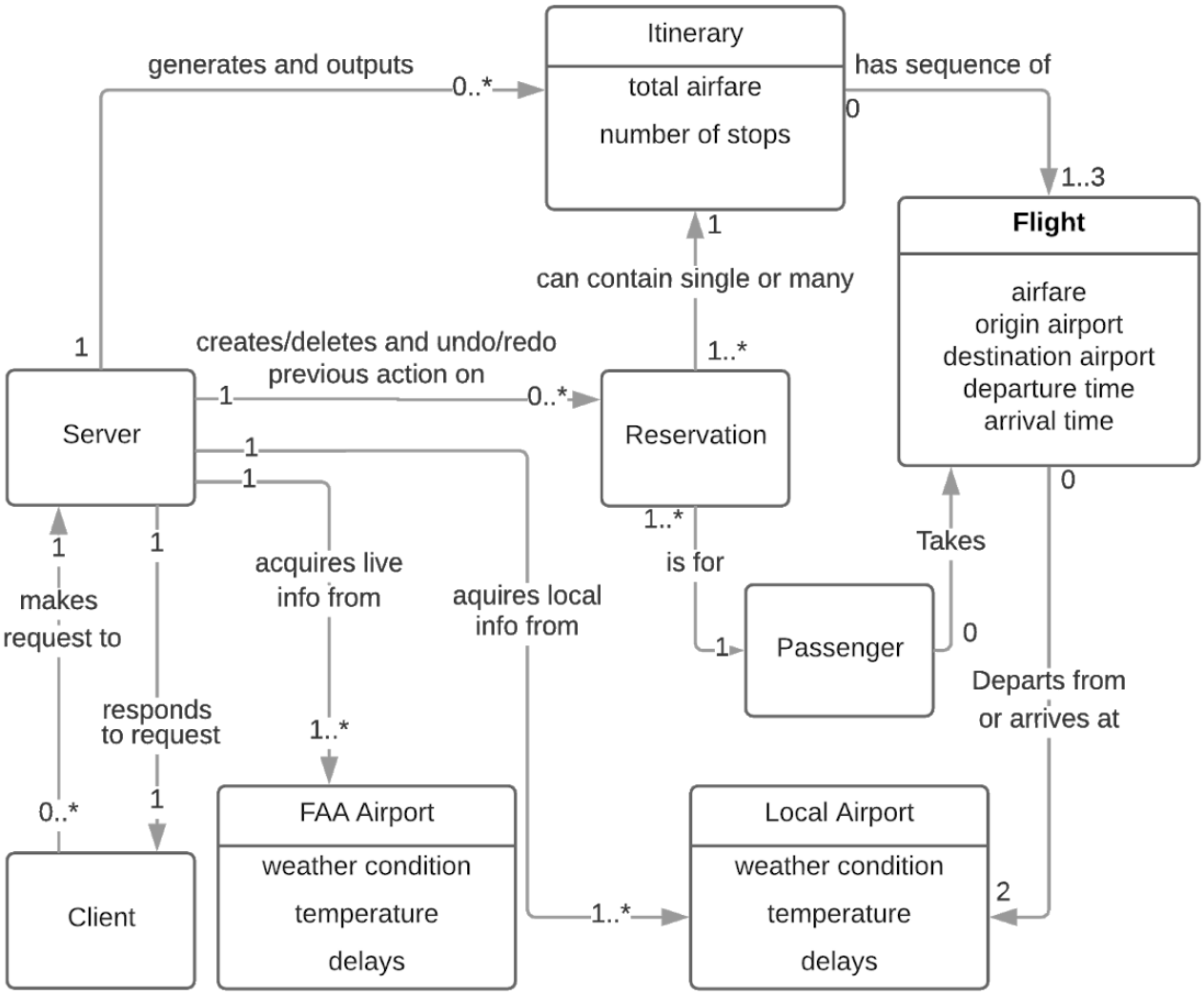
Design Documentation Release 2
Prepared by Strategic Slackers:

- Adam Del Rosso avd5772@rit.edu
- Meet Patel msp2248@rit.edu
- Shawn Struble srs1640@rit.edu
- Tom Amaral txa2269@rit.edu

Summary

Airline Flight Reservation Server is a project commissioned by the Treetop Airways. The application is responsible for providing flight information; making, storing and deleting passenger reservations; providing airport information from either a local source or the Federal Aviation Administration web API; and allows for multiple concurrent client connections to the database.

Application Domain



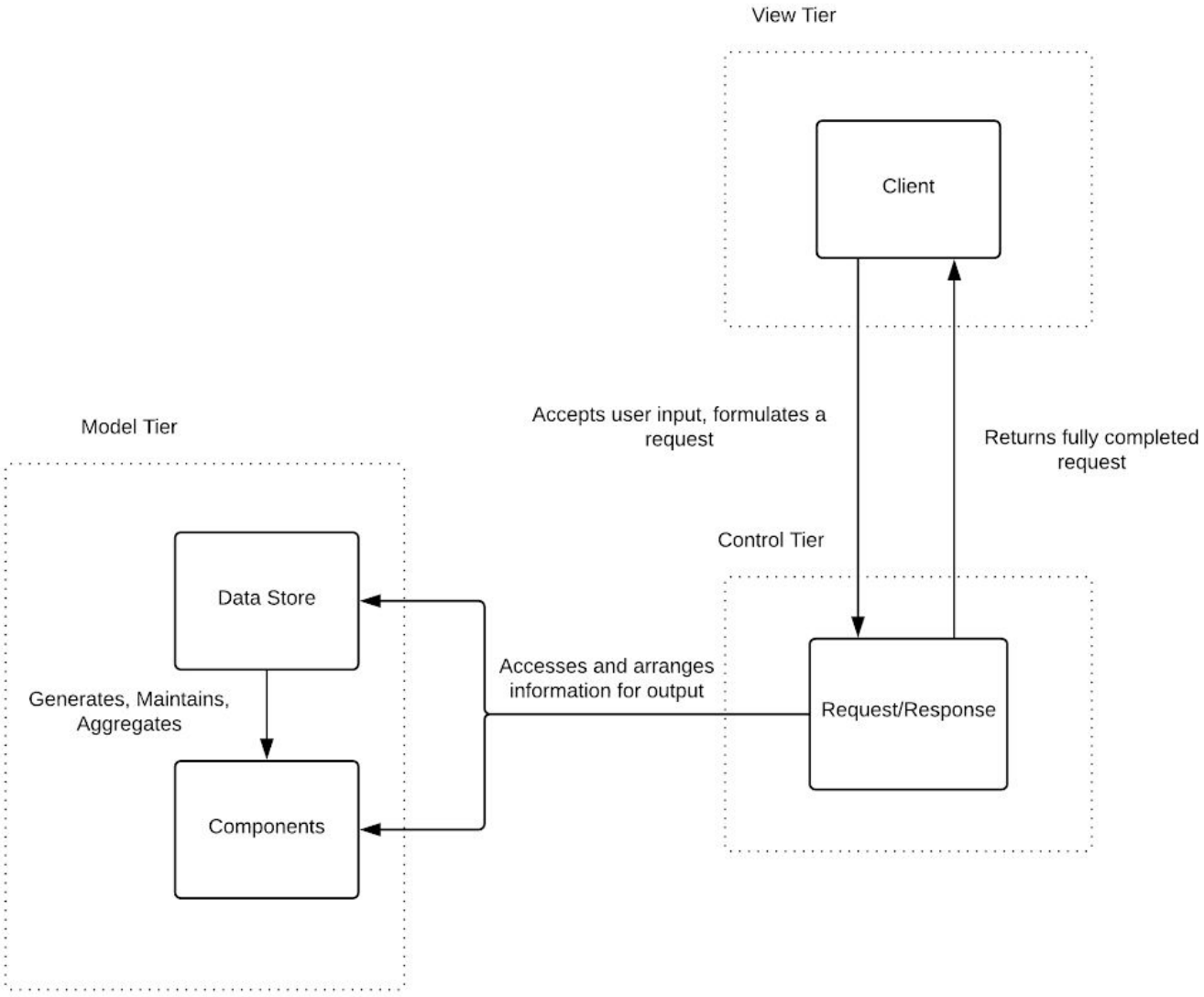
Overview of Major Domain Areas

Figure 1 shows the domain model for the application. Within this domain are the following elements

- Client: The terminal interface through which a user accesses the AFRS api.
- Server: A high level representation of the control and model tier.

- Itinerary: An itinerary is a collection of flights with a specified sequence.
- Reservation: An itinerary for a passenger.
- Flight: An individual flight has a specific cost and contains an origin and destination, along with departure and arrival times.
- Local Airport: The airports used by TTA, which have weather conditions and temperature. This airport data is stored locally.
- FAA Airport: An api which can be used to look up live information on major airports throughout the country. Reports on current delays, temperature and weather conditions.
- Passenger: A passenger may have a reservation that contains his flights in the form of an itinerary.

System Architecture



Our system is separated by tiers and further organized by subsystems. The Client subsystem makes up the entire view tier. This subsystem provides a graphical user interface to use with the system. It also

includes an optional text based terminal to operate the system with. It is responsible for accepting user input and passing this off to the Request/Response subsystem in the Control tier. The input string is taken in by the RequestHandler class in the Request/Response subsystem. From there, the input is further processed through the use of commands specific to the input type. Information necessary to complete the request is then retrieved from the Data Store subsystem in the Model tier. The Data Store subsystem is responsible for aggregating all the entities within the Components subsystem. Data Store also holds behavior to access and manipulate the data stored within. Once the information needed for the request is received, the Request/Response subsystem formats the information and returns it to the Client. This is then displayed to the user and the cycle is completed.

Release 2 Design Updates

Updates from release 1:

One major design decision that was made during the creation of release 1 was on startup to create all itineraries possible based on the flights that were loaded in to the FlightStore. Storage of reservations was created by mapping passenger names to hashes of their itineraries. After further analysis of the this in the context of our system, this was concluded to be excessive. This has since been updated in a number of ways. Itineraries are now created only when a client requests them with a specific origin and destination.

Furthermore, the storage of reservations is done by mapping passenger names to a list which holds their itineraries. In release one, it was thought that offline storage would be done by storing the hashes of itineraries next to the passenger's name. This was updated by simply storing a passenger name with a list of their flight numbers in succession aligning to corresponding succession of flight numbers in an itinerary. During system startup, a reservation file is read from which passes flight numbers to the itinerary store which returns an itinerary from those numbers. The reservation store then maps the passengers name to those itineraries. As of release one, functionality with persistent storage of itineraries across system shutdowns was not functional. This functionality is now fully operational in release two. In addition, functionality of the reservation and retrieve commands has been restored. With these changes completed, all of the functional requirements and non functional requirements associated with release one requirements should be complete.

Updates from release 2:

There are three major design changes associated with release 2. One decision was made to fulfill the non-functional requirements outlined by 4-C which details the use of not only locally stored airport information for the airport command but also an external FAA api. The requirements state for the api to do live lookups of current weather,delays, and names for major airports around the USA. To account for the new requirements for the airport command, the proxy pattern was utilized. An FAA airports class was created that has the ability to do the operations outlined in the requirements to look up temperature, delays, weather, and airport names. The proxy pattern was used to locally get airport names from pre-existing airport data. Moreover, the requirements detailed the ability to change

between the local database and the FAA api. This resulted in a new command, server, which was added to the request/response subsystem. The use of Factory and command in release one made adding this command and all additional ones a very simple process. The server command allows clients to choose between which airport information store they will pull data from. This adheres to the requirements outlined by non-functional requirement 4-A.

The addition of multiple concurrent clients connected to the system results in many portions of the system needing to know when new clients are connected, their individual id, and also when they disconnect. This functionality was accomplished by implementing a push observer pattern. The use of observer pattern allowed system components such as ServerStore, Airport(Model), and InfoQueryStore to know about the activity of client connections without knowing about the manager of the clients. There is one observer, RequestHandler, that does know about the ID manager. This is because the RequestHandler handles connections and disconnections from clients, and in both cases the ID manager needs to be informed.

The final design update resulted from a new requirement that specifies certain commands having stateful behavior to be undoable and redoable. This adheres to the functional requirement specified in 5-D. A new manager, the UndoRedoManager, gets valids commands from the request handler. The UndoRedoManager, is responsible for checking whether or not the command is stateful. Stateful commands implement from another interface which also implement the command interface. The UndoRedeoManager checks if the command passed is one of these commands. These commands include reserve and delete. If one of these commands is found, it is added to a stack of commands which can be undone after being executed. Normal commands will simply have there execute method called. If an undo or redo command is found by the UndoRedoManager, assuming a stateful command was previously done by the client, the top of the respective stacked is returned and and the appropriate operation to either undo or redo the command is done.

Subsystems

This section provides detailed design for each subsystem described in the system architecture.

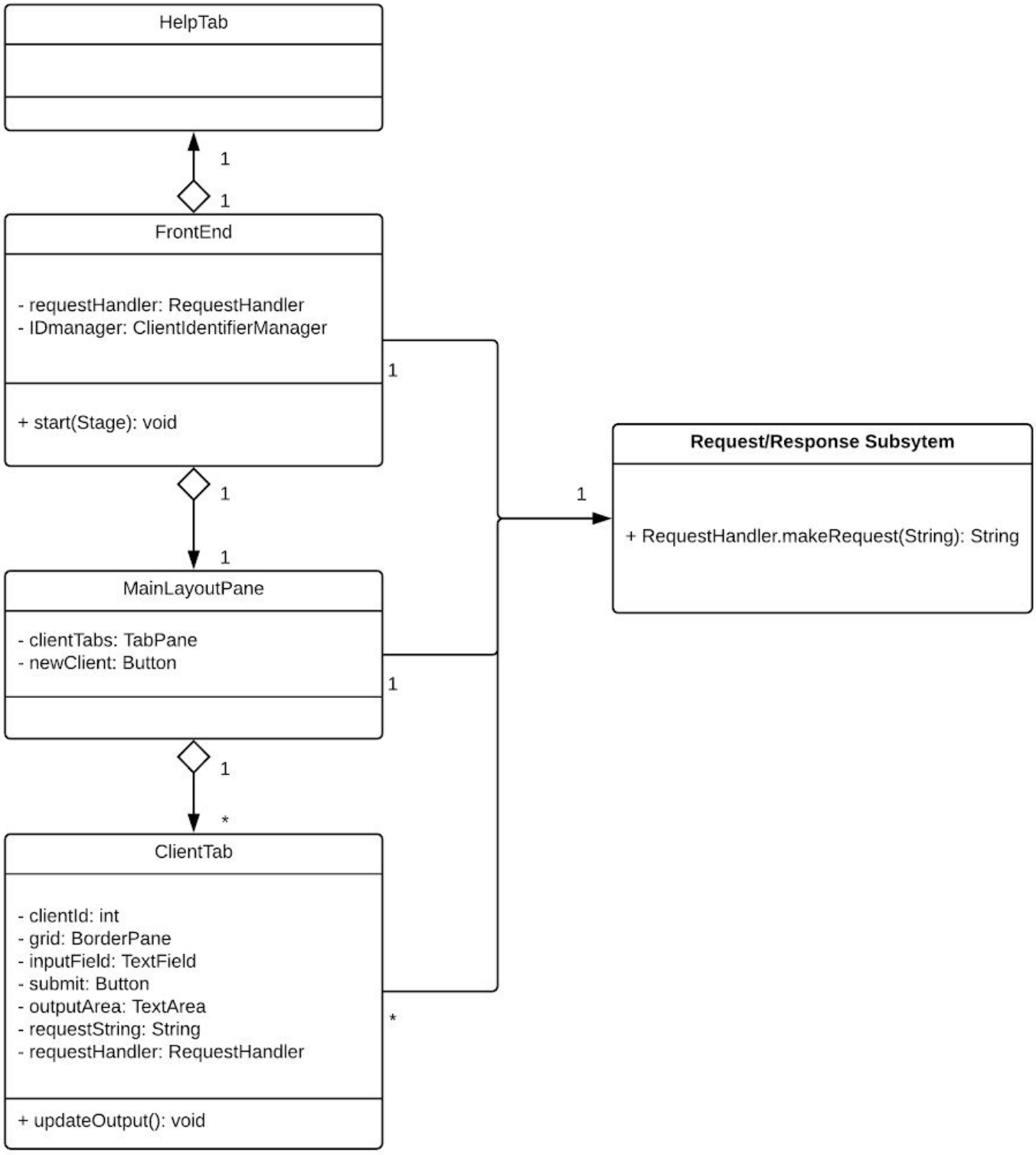
Client

The Client subsystem is responsible for receiving user input from the user, relaying the input to the Request/Response subsystem, and outputting the responses from the Request/Response system back to the user. The Client subsystem consists of 4 classes: FrontEnd, MainLayoutPane, ClientTab, and HelpTab. These classes can be seen below in **Fig 2-1**. With the exception of FrontEnd, the other components are simply customized versions of JavaFX elements in order to break up the responsibilities of FrontEnd. HelpTab provides a landing page for when the program is first loaded. FrontEnd is the starting point of the entire AFRS. It instantiates the RequestHandler which is then passed through MainLayoutPane to each ClientTab that is created. After the components are initialized, they are added to the Stage and the GUI is revealed to the user.

Requirements Covered

The Client subsystem fulfills the R2 requirement of being graphical in nature. Individual client connections are separated by corresponding tabs for each connection. Each client tab automatically connects to the RequestHandler on instantiation and receives their client ID from the response.

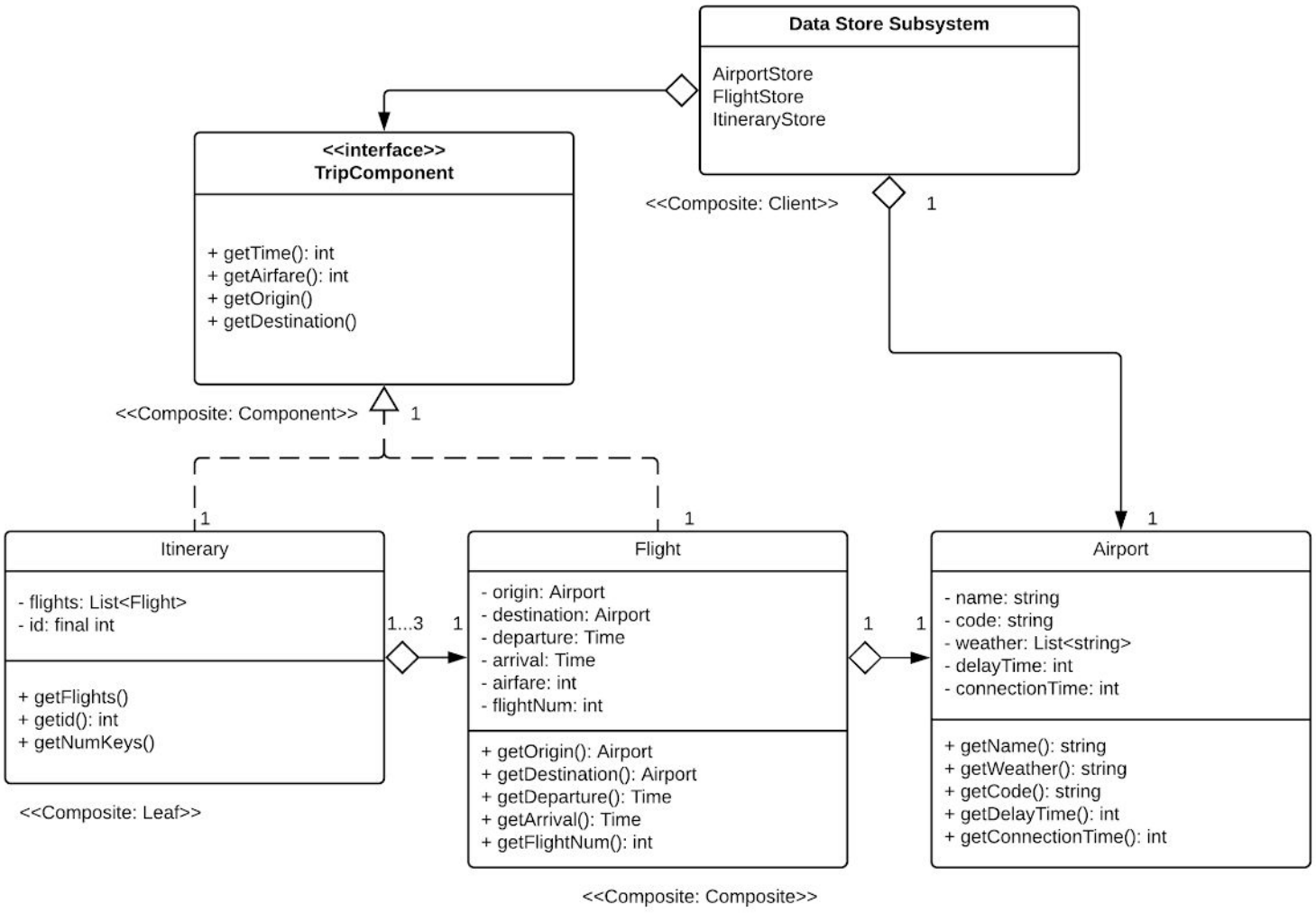
UML Fig 2-1


HelpTab
1
1
FrontEnd
- requestHandler: RequestHandler
- IDmanager: ClientIdentifierManager
+ start(Stage): void
1
1
1
MainLayoutPane
- clientTabs: TabPane
- newClient: Button
1
1
*
ClientTab
- clientId: int
- grid: BorderPane
- inputField: TextField
- submit: Button
- outputArea: TextArea
- requestString: String
- requestHandler: RequestHandler
+ updateOutput(): void
*
1
Request/Response Subsytem
+ RequestHandler.makeRequest(String): String

Components

The Component subsystem contains the classes Flight, Airport, Itinerary and the TripComponent interface. The role of this system is to contain all of the data objects that are stored in the data stores inside the data storage subsystem. The objects created in this subsystem contain data that is parsed to them from system files.

Fig. 3-1



Requirements Covered

The Component Subsystem doesn't fulfill any specific requirements, but it represents the entities of the project statement and provides the foundation on which the remainder of the system operates on.

Design Patterns:

Fig. 3-2

Name: Trip Comonents		**GoF pattern: Composite**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
TripComponent	Component Interface	TripComponent Interface is the interface that is implemented by all Composite nodes (Itinerary) and Leaf nodes (Flight). This is used so you can treat all of the nodes the same in order to traverse through the tree structure recursively.
Itinerary	Composite	Itenerary is the composite because it has the functionality of adding and removing children. These children can either be other Itineraries or flights.
Flight	Leaf	The flight class is the leaf node because it does not have the ability to maintain children.
ItineraryStore	Client	Takes a TripComponent and makes an Itinerary from it.
Deviations from the standard pattern: There are no child related methods inside of the TripComponent interface, so they wouldn't have to be nulled out when called by a leaf node.		

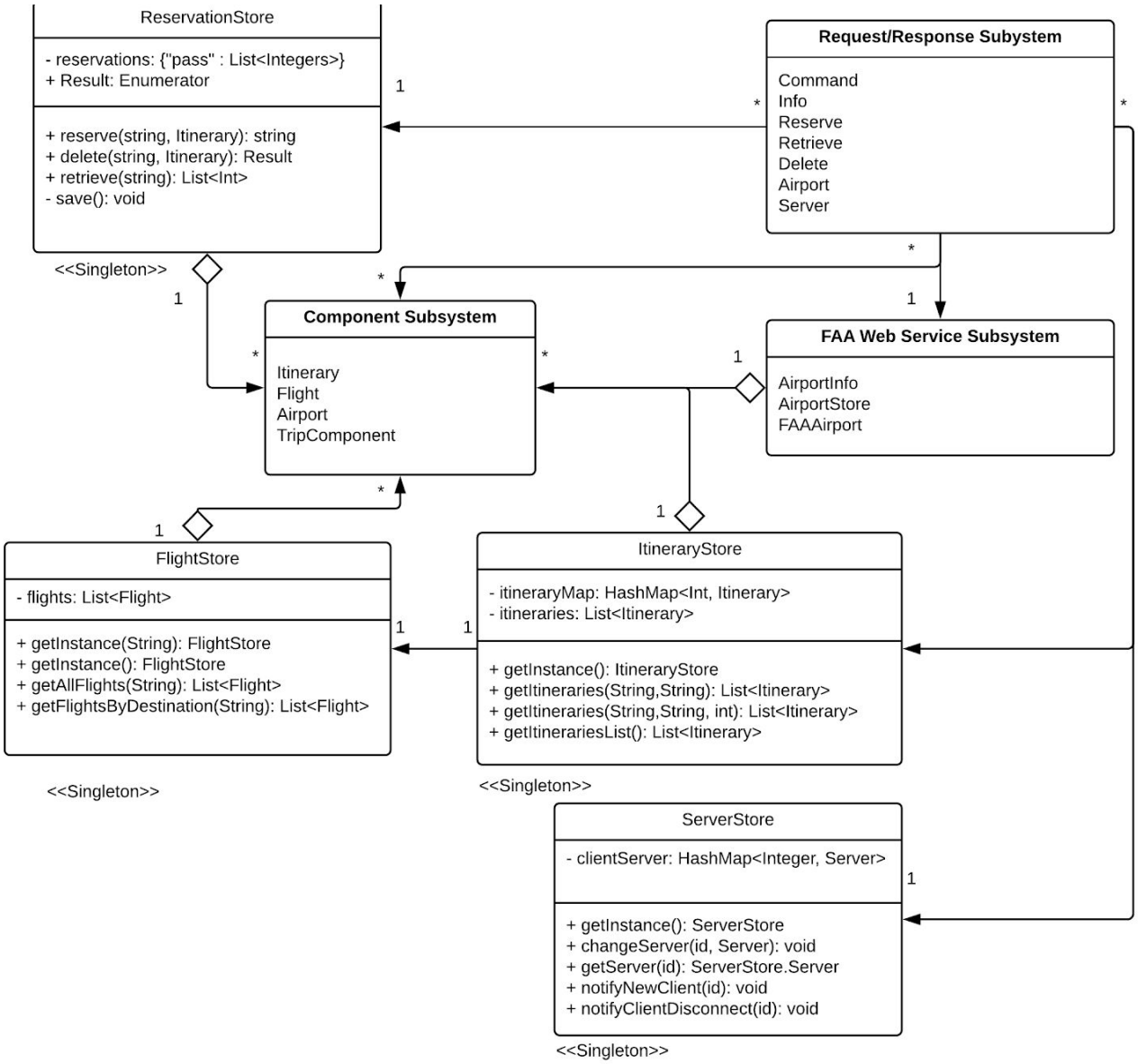
Major Design Decisions:

The requirements stated that an itinerary should maintain the total airfare and total price of all of the flights in the itinerary. The composite pattern, shown in **Figure 3-2**, is very applicable to this requirement because it allows the itinerary and flight objects to be presented in a tree like structure to represent a part-whole hierarchy where you can treat both objects uniformly. Treating them uniformly is important because this allows recursively traversing the tree in order to add up all of the airfare and price values of each flight node. In order to do this, we had an interface called TripComponent which was implemented by both itineraries and flights.

Data Store

The Data Store Subsystem, seen below in **Fig 4-1**, is a collection of singleton database classes that manage and provide domain information to the rest of the system. The different data areas include reservations, airports, flights, and itineraries based on the flights read in. The data stores contain aggregated data all of the data components which are used by the Request/Response subsystem, and contain all the necessary behavior to retrieve information from their components. This choice was made to adhere to the idea that functionality should follow data. It became obvious to the team that the stores should have the capability to query their own data. When the system initiates the data stores are responsible for loading the given data files and generating all the objects of the Component Subsystem. Additionally, the Data Store contains the FAA Web Service Subsystem within it.

Fig. 4-1



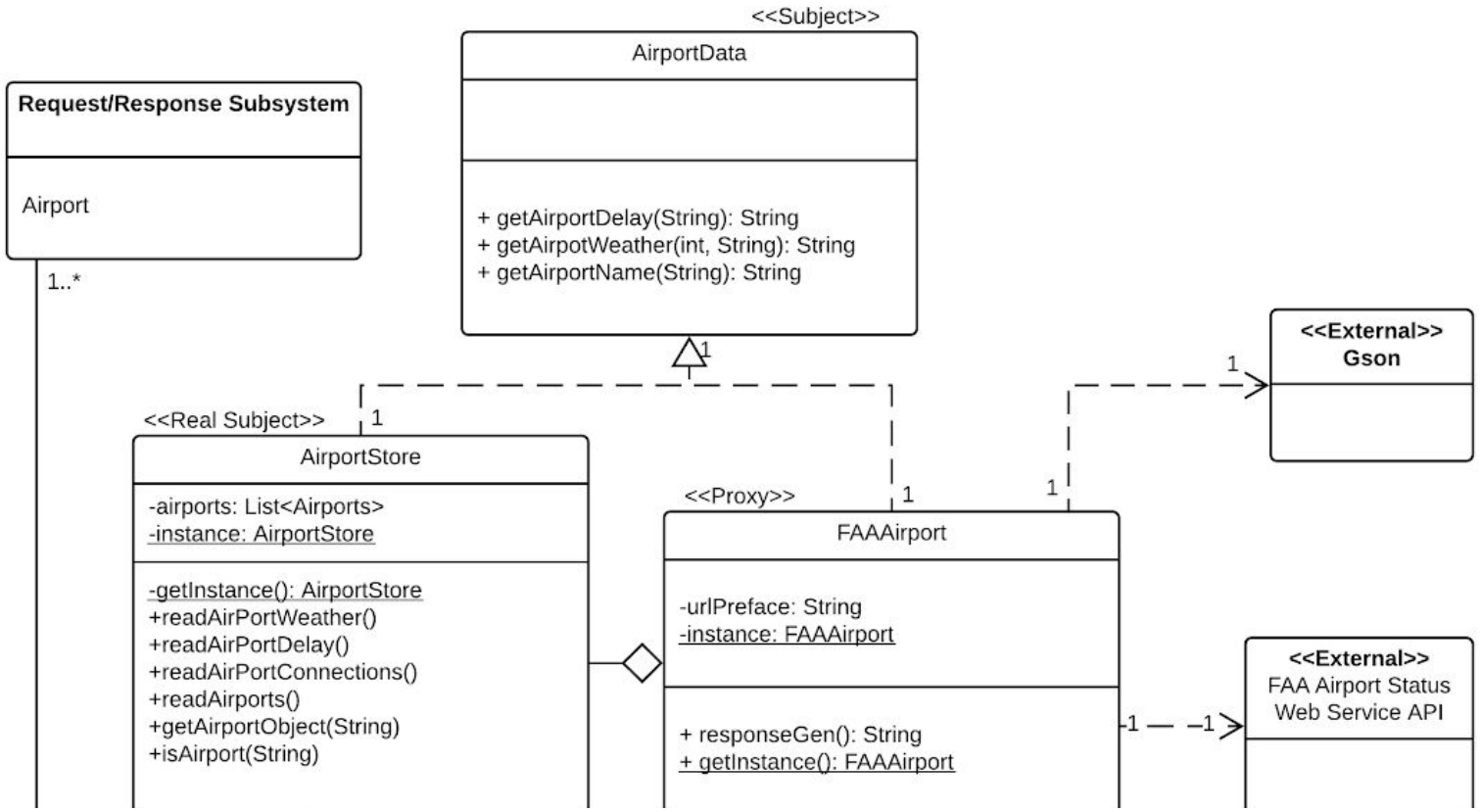
Requirements Covered:

The Data Store subsystem maintains all the information needed to complete the requests given to the Request/Response Subsystem. This includes itineraries, airport information, airfare for flights, and reservations made.

FAA Web Service

The FAA Web Service Subsystem is responsible for allowing the client to choose between receiving airport information from either local files, or a remote web service. To achieve this in R2, an AirportData interface was added. Both the previous AirportStore from R1 and a new class FAAAirport implement this interface. FAAAirport acts as a Proxy around AirportStore in order to use its getAirportName() behavior, while adding its own to retrieve the remote data from the web service. This is accomplished with Google's GSON library and the use of HTTP Json requests to the FAA web service.

Fig. 5-1



Requirements Covered:

The FAA Web Service Subsystem fulfills the requirement of providing the option of receiving airport information from either local files or a remote webservice. Additionally, it also holds the AirportStore, which loads up all the airport info from the local files provided. AirportStore and FAAAirport provide all the airport information needed within the system to complete airport requests.

Design Patterns:

Fig. 5-2

Name: Airport Info		**GoF pattern: Proxy**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
AirportData	Subject	AirportData provides the behavior to get the airport name, weather, and delay to its subclasses.
FAAAirport	Proxy	Uses its internal instance of AirportStore to call AirportStore's method to access airport names. Provides its own behavior for accessing the remaining information needed from the FAA webservice.
AirportStore	Real Subject	Implements the behavior outlined by AirportData to fetch information from the local airport information files.
Deviations from the standard pattern: None		

Major Design Decisions:

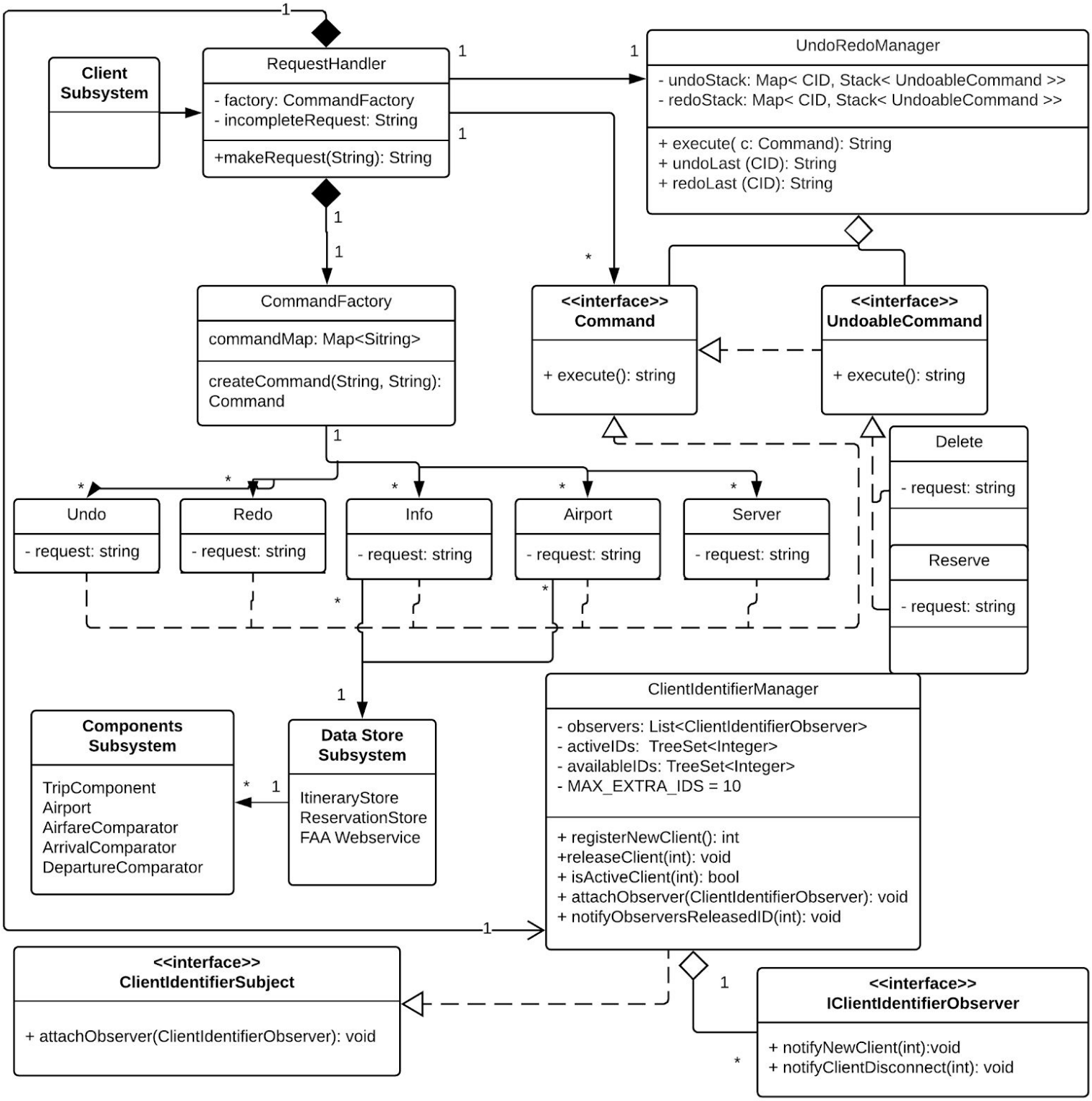
The use of Proxy was chosen here because the airport names are not changing between the local and remote data. The classes of FAAAirport and Airport also share the same required return data for airports which allows an interface to be implemented from. Therefore, it was decided that the proxy pattern was appropriate and the FAAAirportStore would contain an instance of AirportStore, and use its getAirportName behavior.

Request / Response

The Request/Response Subsystem is responsible for receiving and responding to clients. The RequestHandler waits for complete input and a valid command keyword from the client. Then the command keyword is passed to the CommandFactory, which creates a new Command and returns it back the the RequestHandler. These commands are based off of the command design pattern. Command classes contain the necessary logic to process client request parameters and carry out command functionality. Due to the singleton nature of data store objects, commands are able to acquire their own model their resources.

Commands are given to the UndoManager once received by the RequestHandler, and in most cases output from the Command is returned back to the Client. An exception to this is the Undo and Redo commands, which when given to the UndoManager trigger an undo or redo operation. The output from the undo or redo operation is what makes it back to the client. The RequestHandler knows nothing about the logic within Commands or the response that the Client expects. The full class diagram can be seen below in **Figure 6-1**.

Fig. 6-1


Client Subsystem
RequestHandler
- factory: CommandFactory
- incompleteRequest: String
+makeRequest(String): String
UndoRedoManager
- undoStack: Map< CID, Stack< UndoableCommand >>
- redoStack: Map< CID, Stack< UndoableCommand >>
+ execute(c: Command): String
+ undoLast (CID): String
+ redoLast (CID): String
CommandFactory
commandMap: Map<Sitring>
createCommand(String, String): Command
<<interface>> Command
+ execute(): string
<<interface>> UndoableCommand
+ execute(): string
Undo
- request: string
Redo
- request: string
Info
- request: string
Airport
- request: string
Server
- request: string
Delete
- request: string
Reserve
- request: string
Components Subsystem
TripComponent
Airport
AirfareComparator
ArrivalComparator
DepartureComparator
Data Store Subsystem
ItineraryStore
ReservationStore
FAA Webservice
ClientIdentifierManager
- observers: List<ClientIdentifierObserver>
- activeIDs: TreeSet<Integer>
- availableIDs: TreeSet<Integer>
- MAX_EXTRA_IDS = 10
+ registerNewClient(): int
+releaseClient(int): void
+ isActiveClient(int): bool
+ attachObserver(ClientIdentifierObserver): void
+ notifyObserversReleasedID(int): void
<<interface>> ClientIdentifierSubject
+ attachObserver(ClientIdentifierObserver): void
<<interface>> IClientIdentifierObserver
+ notifyNewClient(int):void
+ notifyClientDisconnect(int): void

Design Patterns:

Fig. 6-2

Name: Command Factory		**GoF pattern: Factory**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
RequestHandler	Client	Passes the input keyword to the CommandFactory so that it can create the appropriate concrete product. Receives the completed command object back.
CommandFactory	Concrete Factory	Given a request keyword, finds the proper command object needed to fulfill the request, and returns an instantiated command back to the RequestHandler.
Command	Abstract Product	Command is an abstract class in our system that outlines the basic behavior that any given command will have. This behavior currently defined is to execute the command.
Info	Concrete Product	This concrete command acts on ItineraryStore in order to obtain requested information about flight itineraries.
Reserve	Concrete Product	This concrete command acts on the ReservationStore to add the requested itinerary to a given passenger.
Retrieve	Concrete Product	Given a name, this command acts on ReservationStore to retrieve an already made reservation with that name.
Delete	Concrete Product	Given a name, origin, and destination, the delete command acts on ReservationStore to delete the matching itinerary from a passengers reservation.
Airport	Concrete Product	Given an airport code, acts on AirportStore to retrieve information on the given airport.
Server	Concrete Product	Changes the airport info service from either local or FAA.
Undo	Concrete Product	Provides a way for the UndoManager to see if there is a request to undo another command. Undoes the latest undoable command.
Redo	Concrete Product	Provides a way for the UndoManager to see if there is a request to redo another command. Redoes the latest redoable command.
Deviations from the standard pattern: Our design blends both the Command and Factory pattern. Additionally, we did not include an Abstract Factory layer, as we will only ever need one type of Factory in this subsystem.		

We agreed that it was acceptable for the RequestHandler to be aware of the concrete type of CommandFactory.

Fig. 6-3

Name: Request Commands		**GoF pattern: Command**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
Command	Abstract Command	Command is an abstract class in our system that outlines the basic behavior that any given command will have. This behavior currently defined is to execute the command.
UndoAble	Abstract Command	Interface holding behavior for all undoable commands in the system.
Info	Concrete Command	This concrete command acts on ItineraryStore in order to obtain requested information about flight intineraries.
Reserve	Concrete Command	This concrete command acts on the ReservationStore to add the requested itinerary to a given passenger.
Retrieve	Concrete Command	Given a name, this command acts on ReservationStore to retrieve an already made reservation with that name.
Delete	Concrete Command	Given a name, origin, and destination, the delete command acts on ReservationStore to delete the matching itinerary from a passengers reservation.
Airport	Concrete Command	Given an airport code, acts on AirportStore to retrieve information on the given airport.
Server	Concrete Product	Changes the airport info service from either local or FAA.
RequestHandler	Invoker	Receives an request string from the Client subsystem. Passes the request keyword to CommandFactory and gets the appropriate command back. Invokes the command, returning the output to the Client subsystem.
CommandFactory	Client	Given a request keyword, computes the proper command object needed to fulfill the request, and returns an instantiated command back to the RequestHandler.
ItineraryStore	Receiver	The database of itineraries. Generates Itinerary combinations based off of info requests.
ReservationStore	Receiver	The database of reservations. Holds all reservations made persisted across start ups.
AirportStore	Receiver	The database of AirportStore. Stores all airports, loaded in from file.

FAAAirport	Receiver	Class that retrieves information from the FAA web service and makes it available to the user request.
Deviations from the standard pattern: Our design blends both the Factory and Command Pattern.		

Fig. 6-4

Name: Client IDs		**GoF pattern: Observer**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ClientIdentiferSubject	Subject	Attaches a new observer to the subject.
ClientIdentiferObserver	Observer	outlines functionality for ClientIdentiferManager to implement. Includes ability to release client ids and notify observers of new client id numbers added.
ClientIdentiferManager	Concrete Subject	Manages client connections. Does so by keeping track of client identification numbers. Notifies observers that a new client identification number has been assigned. Notifies observers that a client has disconnected.
InfoQueryStore	Concrete Observer	Oberserves ClientIdentiferManager to know when client Id's have been added to or deleted from the system. Uses client Id's to map them to most recent info command used by the client.
ServerStore	Concrete Observer	Oberserves ClientIdentiferManager to know when client Id's have been added to or deleted from the system. Maps client Id's to the server that they are currently connected to, either faa or local.
Airport	Concrete Observer	Oberserves ClientIdentiferManager to know when client Id's have been added to or deleted from the system. Maps client Id's to current local airport weather which will be iterated through each time it is called from a specific client.
Deviations from the standard pattern: None		

Major Design Decisions:

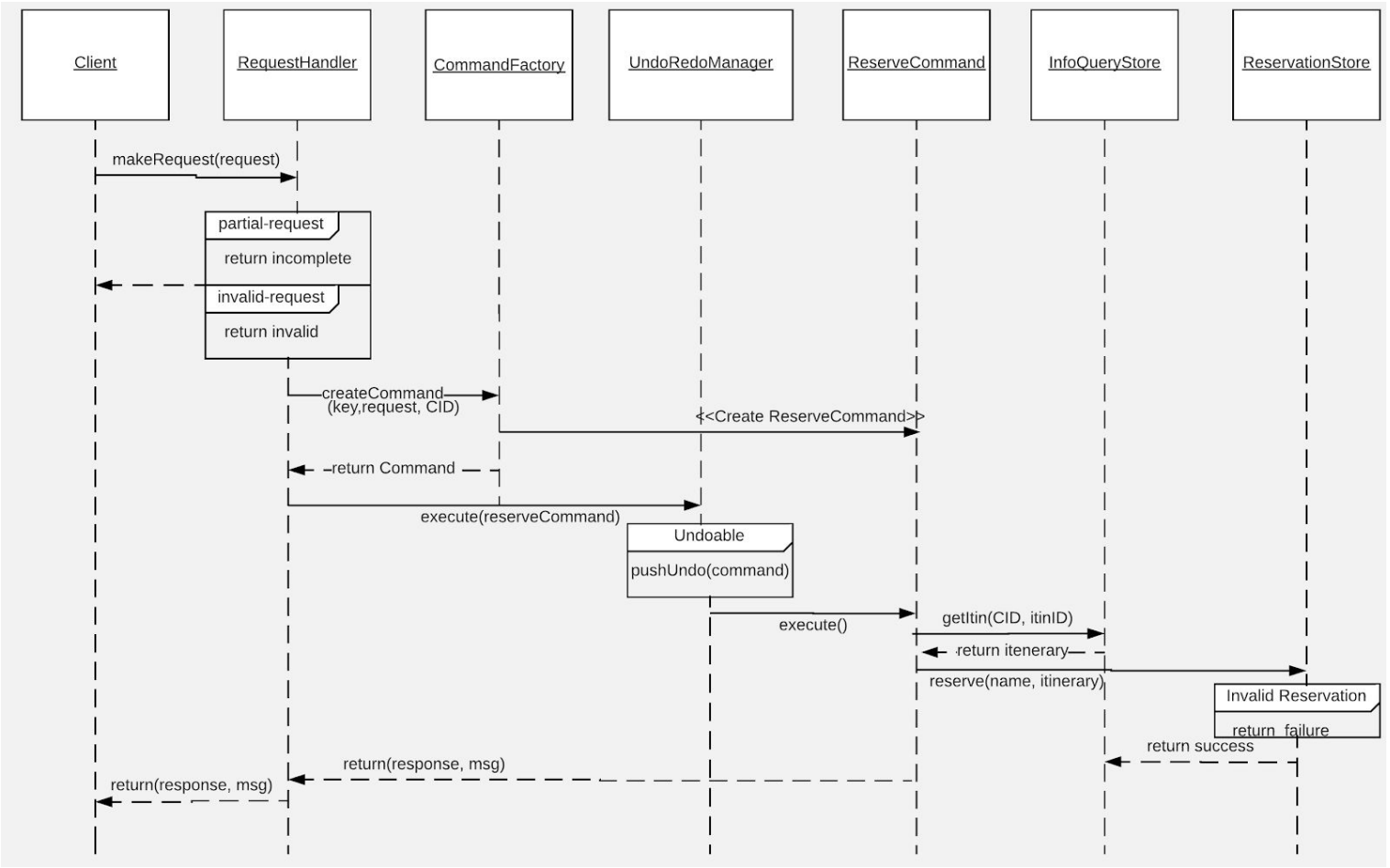
The observer pattern was used to support the multiple client requirement. Rather than the manager of the client IDs knowing which pieces of the system need to be notified of new or disconnected clients, objects can be attached as observers to receive push notifications. This results in a lower coupling between this ID manager and the rest of the system, and allows new objects which need client ID information to be added in the future.

To support the undoing and redoing of commands, a new UndoableCommand interface was created which extends the original Command interface. The UndoManager, which is given all commands to execute, will automatically save commands that are implementing this new interface for later undo or redo operations. This allows for the easy addition of new undoable commands in the future, as opposed to the UndoManager needing to know about each concrete command that should be undoable.

Release 2 Sequence Diagrams

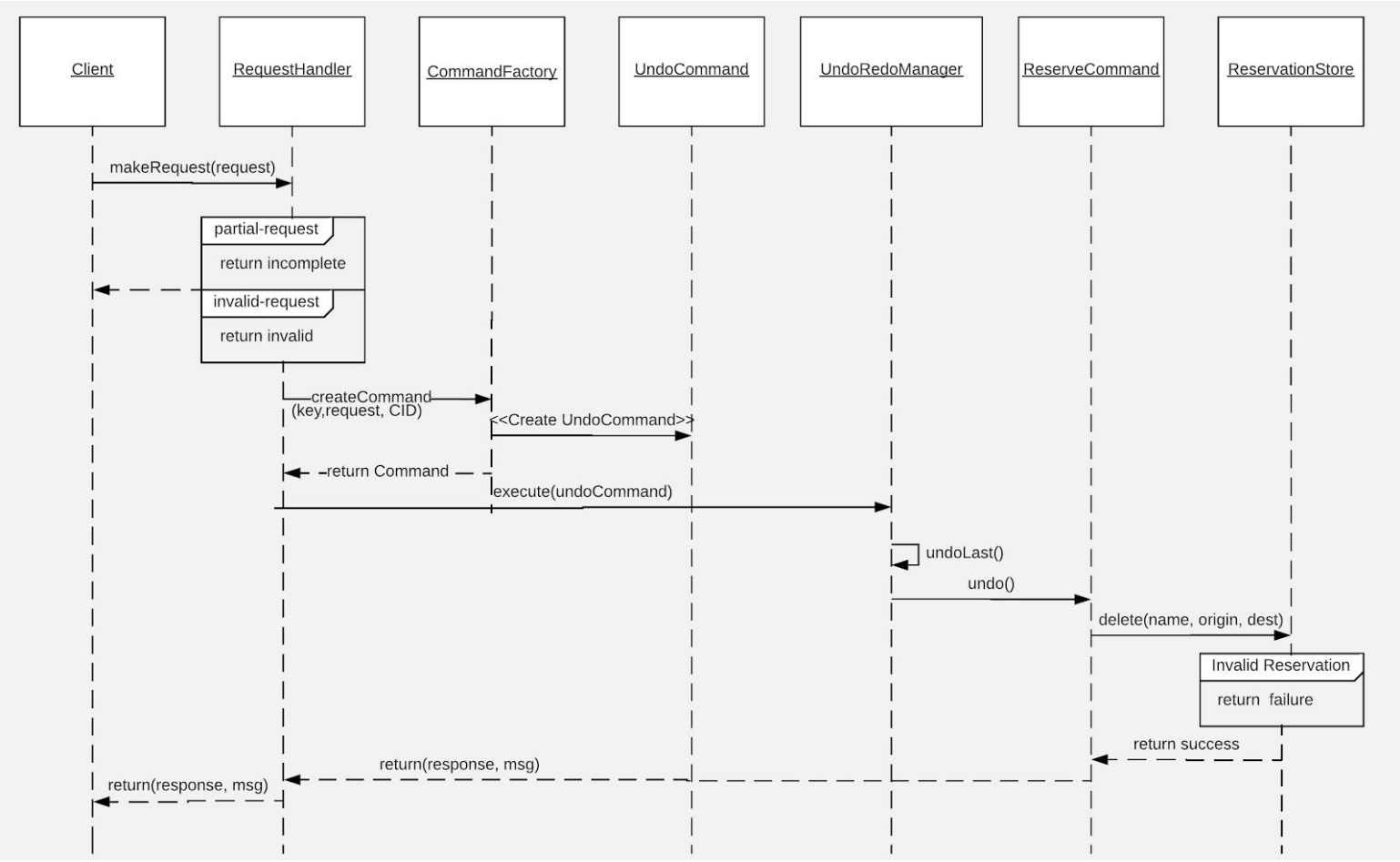
Adding a Reservation

This sequence diagram is used to highlight how the undo-redo subsystem handles adding a reservation. In this process, the client first makes a request that is passed onto its RequestHandler. Then, the RequestHandler checks if the request string is partial or invalid, which if it is, an error message is sent back to the client. Else, the RequestHandler creates the Reserve command from the commandFactory using a keyword from the request string and then passes that command to its UndoRedoManager. Then the UndoRedoManager executes the command, in which it first checks if the command implements UndoableCommand. Because Reserve is a state changing command, it implements UndoableCommand, which pushes it onto the UndoStack of the UndoRedoManager. Next, when the reserve command is executed, it adds a reservation to the ReservationStore singleton class by calling add() using name, origin, and destination and this returns failure if the reservation that is being added is invalid, or success, if the reservation is successfully added. Finally, the requestHandler creates a response message and passes it to the client.



Undoing a Reservation

This sequence diagram is used to highlight how the undo-redo subsystem handles undoing a reservation. In this process, the client first makes a request that is passed onto its RequestHandler. Then, the RequestHandler checks if the request string is partial or invalid. If it is, an error message is sent back to the client. If not, the RequestHandler creates the Undo command from the commandFactory using a keyword from the request string and then passes that command to its UndoRedoManager. Then, the UndoRedoManager executes the Undo command. This calls the private method undoLast() in the UndoRedoManager, which calls undo() on the last state changing command that was instantiated. In this case it would be a Reserve command. When undo() is called on the Reserve command, reserve does the opposite of what its execute() method would do, which would be to delete a reservation. To do this, it calls delete() on the ReservationStore singleton to delete a reservation using name, origin and destination. This method returns failure if the reservation that is being deleted is invalid, or success, if the reservation is successfully deleted. Finally, the requestHandler creates a response message and passes it to the client.



Appendix

Class: `Airport`	
Responsibilities: Creates an Airport object which is used as a arrival land destination attribute for flight objects	
Collaborators:	
Users: None	**Used by:** Flight
Author: Meet Patel	

Class: `Airport`	
Responsibilities: Retrieves information about the requested airport and returns it to RequestHandler.	
Collaborators:	
Users: Airport, AirportStore	**Used by:** CommandFactory, RequestHandler
Author: Adam Del Rosso	

Class: `AirportStore`	
Responsibilities: Loads in all airports from file as Airport objects and aggregates them for use by the Request/Response subsystem. Provides behavior for accessing its information.	
Collaborators:	
Users: Airport	**Used by:** Airport, CommandFactory
Author: Shawn Struble	

Class: `Command`	
Responsibilities: Abstract class outlining the behavior of any given Command subclass. The currently defined behavior is to execute the command.	

Collaborators:	
Users: None	**Used by:** RequestHandler, UndoRedoManager, CommandFactory
Author: Adam Del Rosso	

Class: `CommandFactory`	
Responsibilities: Takes in a request keyword from RequestHandler. Decides which command object will fulfill the request. Instantiates the command and returns it to RequestHandler.	
Collaborators:	
Users: Info, Reserve, Retrieve, Delete, Airport	**Used by:** RequestHandler
Author: Adam Del Rosso	

Class: `Delete`	
Responsibilities: Given a string with a passenger name, origin and destination airport, deletes the matching reservation from the ReservationStore if the string is valid.	
Collaborators:	
Users: ReservationStore	**Used by:** CommandFactory
Author: Adam Del Rosso	

Class: `Flight`	
Responsibilities: Maintain a flight object which connects 2 airport objects, and computes the time to get from one to another. It also implements the TripComponent interface to be a Leaf in the Composite pattern.	
Collaborators:	
Users: Airport, TripComponent	**Used by:** Itinerary, FlightStore

Author: Meet Patel	

Class: `FlightStore`	
Responsibilities: Read flight information from a file. Generate Flight objects from this information. Aggregate the Flight objects for use by ItineraryStore.	
Collaborators:	
Users: Flight	**Used by:** ItineraryStore
Author: Shawn Struble	

Class: `Info`	
Responsibilities: Retrieves a set of itineraries from the ItineraryStore based on the given request parameters if the request is valid. Returns an error otherwise.	
Collaborators:	
Users: ItineraryStore, TripComponent, Airport	**Used by:** CommandFactory
Author: Adam Del Rosso	

Class: `Itinerary`	
Responsibilities: Contains the details for one trip, which includes 1-3 Flights. It also implements the TripComponent interface to be a composite in the Composite pattern.	
Collaborators: None	
Users: Flight, TripComponent	**Used by:** ReservationStore, ItineraryStore
Author: Meet Patel	

Class: `ItineraryStore`	
Responsibilities: Generates all Itinerary	

objects on start-up. Aggregates these and provides information for accessing	
Collaborators:	
Users: Itinerary, FlightStore, Flight, TripComponent, AirportStore, Airport	**Used by:** Info, CommandFactory
Author: Shawn Struble	

Class: `RequestHandler`	
Responsibilities: Receives an request string from the Client subsystem. Sends the request keyword - if there is one - to the CommandFactory. Invokes the returned Command object, returns the output of the Command to the Client for display.	
Collaborators:	
Users: CommandFactory	**Used by:** Client
Author: Adam Del Rosso	

Class: `ReservationStore`	
Responsibilities: Maps a passenger name to a list of Itineraries. Provides behavior for manipulating and retrieving its information.	
Collaborators:	
Users: Itinerary	**Used by:** CommandFactory, Reserve, Retrieve, Delete
Author: Shawn Struble	

Class: `Reserve`	
Responsibilities: Given request parameters of passenger name and itinerary ID, creates a new reservation in the ReservationStore.	
Collaborators:	
Users: ReservationStore, Itinerary	**Used by:** CommandFactory
Author: Adam Del Rosso	

Class: `Retrieve`	
Responsibilities: Given an input string, accesses ReservationStore and returns a list of itineraries matching the given passenger name. Returns an error if invalid.	
Collaborators:	
Users: ReservationStore, Itinerary	**Used by:** CommandFactory
Author: Adam Del Rosso	

Class: `TripComponent`	
Responsibilities: Interface that is implemented by Flight and Itinerary. Provides standard information retrieval behavior for subclasses.	
Collaborators:	
Users: None	**Used by:** Flight Itinerary
Author: Meet Patel	

Class: `UndoRedoManager`	
Responsibilities: Parses and executes all commands, and aggregates state changing commands in order to undo or redo them. It is used by a RequestHandler	
Collaborators:	
Users: UndoableCommand	**Used by:** RequestHandler
Author: Adam Del Rosso	

Class: `UndoableCommand`	

Responsibilities: Interface that is an extension of the Command interface. It is implemented by state changing commands such as Reserve and Delete. This interface adds undo() and redo() methods in addition to the execute() method in the Command interface. These methods undo or redo what the execute() method does.	
Collaborators:	
Users: Command	**Used by:** UndoRedoManager, CommandFactory
Author: Adam Del Rosso	

Class: `UndoCommand`	
Responsibilities: The UndoCommand is a command that implements the Command interface. It is used so it can be executed by the UndoRedoManager to call the private method of the UndoRedoManager undoLast(), to undo the last state changing command.	
Collaborators:	
Users: Command	**Used by:** UndoRedoManager, CommandFactory
Author: Adam Del Rosso	

Class: `RedoCommand`	
Responsibilities: The RedoCommand is a command that implements the Command interface. It is used so it can be executed by the UndoRedoManager to call the private method of the UndoRedoManager redoLast(), to redo the last state changing command that was undid.	
Collaborators:	
Users: Command	**Used by:** UndoRedoManager, CommandFactory

Author: Adam Del Rosso	

Class: `ClientIdentifierManager`	
Responsibilities: The manager gives out IDs to represent clients connected to the system. It also provides push notifications to observers about new clients and clients which have disconnected.	
Collaborators:	
Users: ClientIdentifierObserver	**Used by:** RequestHandler
Author: Adam Del Rosso	

Class: `ClientIdentifierSubject`	
Responsibilities: Interface defining the attach method to add observers to the ClientIdentifierManager.	
Collaborators:	
Users: None	**Used by:** Airport, UndoManager, Server, InfoQueryStore
Author: Adam Del Rosso	

Class: `ClientIdentifierObserver`	
Responsibilities: Interface defining the push methods used by the ClientIdentifierManager to notify observers.	
Collaborators:	
Users: None	**Used by:** ClientIdentifierManager, Airport, UndoManager, Server, InfoQueryStore
Author: Adam Del Rosso	

Class: `Server`	

Responsibilities: Server Command implements the Command interface. When execute() is called, it toggles if the Airport Data is coming from the FAA web service or locally.	
Collaborators:	
Users: Command	**Used by:** CommandFactory
Author: Adam Del Rosso	

Class: `InfoQueryStore`	
Responsibilities: Stores the last set of itineraries gathered from an info command. This allows the client to reserve based on the output identifier number.	
Collaborators:	
Users: None	**Used by:** Info, Reserve
Author: Adam Del Rosso	

Class: `AirportData`	
Responsibilities: Interface defining methods which must be implemented by providers of airport information.	
Collaborators:	
Users: None	**Used by:** AirportStore, FAAAirport, Command\Airport
Author: Shawn Struble	

Class: `FAAAirport`	
Responsibilities: Gathers airport information through the FAA web service.	
Collaborators:	

Users: None	**Used by:** Command/Airport
Author: Shawn Struble	

Class: `ServerStore`	
Responsibilities: Maps Client IDs to an enum of either LOCAL or FAA. Responsible for observing changes in client connections and maintaining client IDs. Manages what client is connected to what airport info service.	
Collaborators:	
Users:	**Used by:** Airport, Server
Author: Adam Del Rosso	

Class: `CLIClient`	
Responsibilities: Main entry point for the command line interface for AFRS. Accepts user input, sends it to the RequestHandler, and outputs the response back.	
Collaborators:	
Users: RequestHandler	**Used by:** None
Author: Tom Amaral	

Class: `ClientTab`	
Responsibilities: Precustomized Tab JavaFX component. Responsible for accepting user input, making requests to RequestHandler, and outputting the responses back.	
Collaborators:	
Users: RequestHandler	**Used by:** MainLayoutPane
Author: Tom Amaral	

Class: `FrontEnd`	

Responsibilities: Main entry point for the GUI of AFRS. Initializes the RequestHandler and the MainLayoutPane, sets the stage, and shows the GUI to the user.	
Collaborators:	
Users: MainLayoutPane, RequestHandler, ClientIdManager	**Used by:** None
Author: Tom Amaral	

Class: `HelpTab`	
Responsibilities: Simply holds a friendly customized splash message for the GUI to display on start up.	
Collaborators:	
Users: None	**Used by:** MainLayoutPane
Author: Tom Amaral	

Class: `MainLayoutPane`	
Responsibilities: This is a customized JavaFX BorderPane that represents the top level node of the GUI. Contains all of the ClientTabs, the HelpTab and formatting for itself.	
Collaborators:	
Users: ClientTab, HelpTab	**Used by:** FrontEnd
Author: Tom Amaral