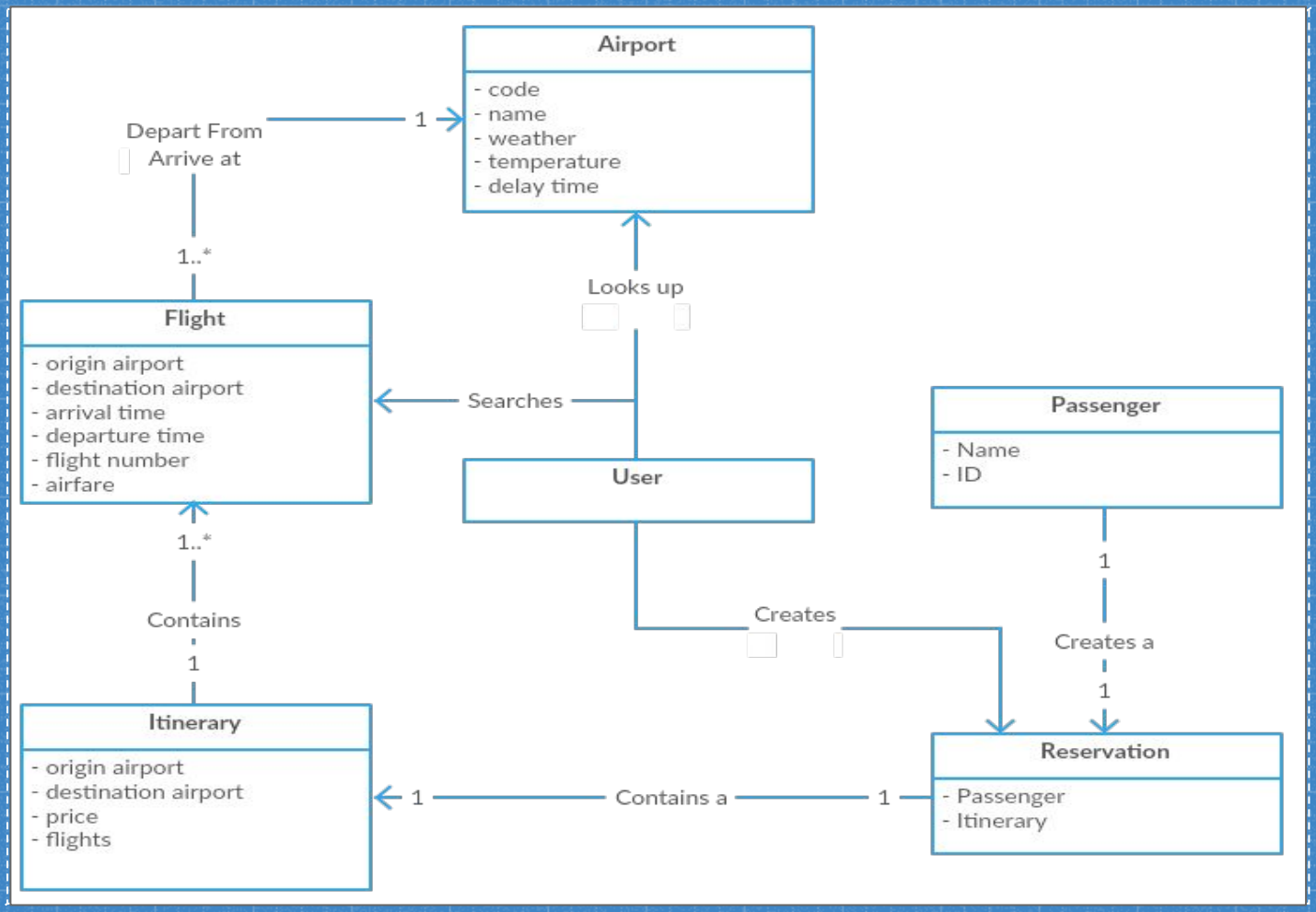
Design Project 1

Airline Flight Reservation Server (AFRS)

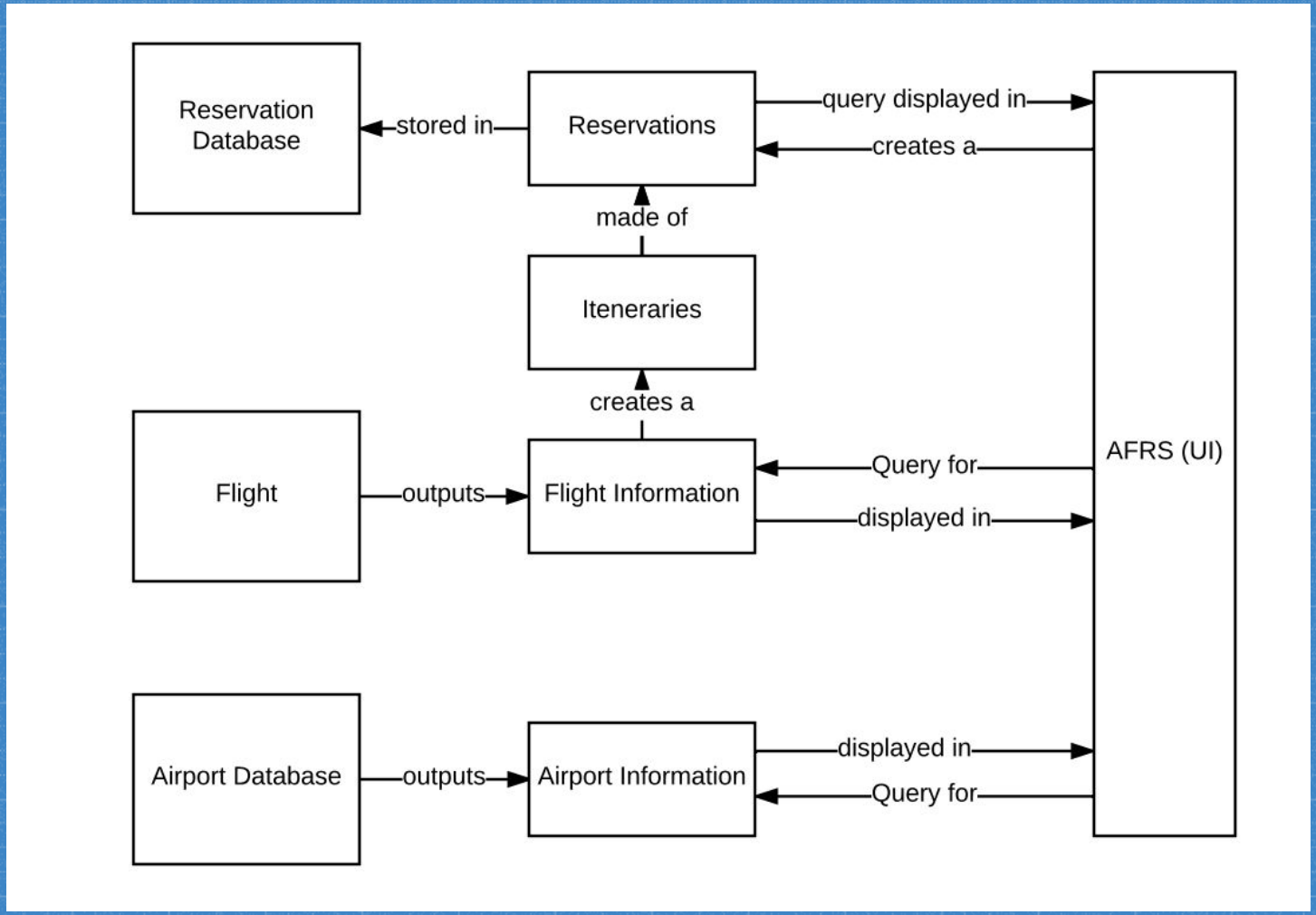
Team 1: Niharika Reddy, Stephen Cook, Joshua Cotton, Moisés Lora

1 System Overview

Domain Model



System Architecture

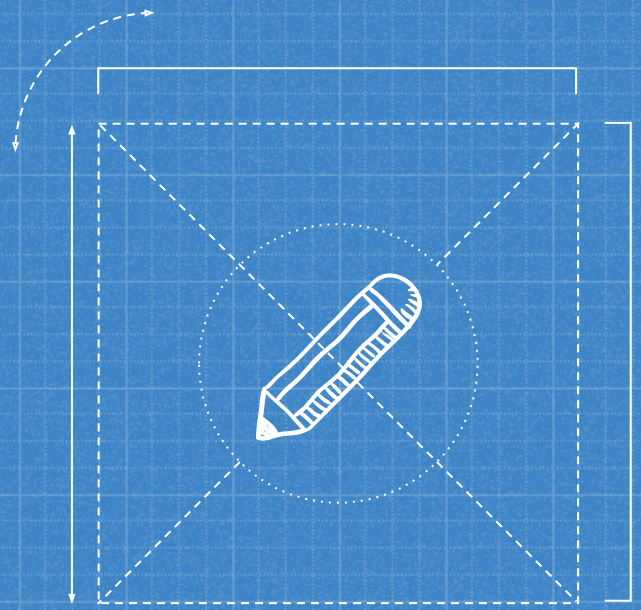


Design Rationale

- Initially designed a large database to hold:
  - Reservations
  - Flight Information
  - Airport Information

- Later, small database or arraylist was used to store each subsystem separately for easier access within the class

2 Subsystems



User Interface Subsystem

Requirements

- Text-based UI
  - ,-delimited, ;-terminated
- Partial requests
- Commands

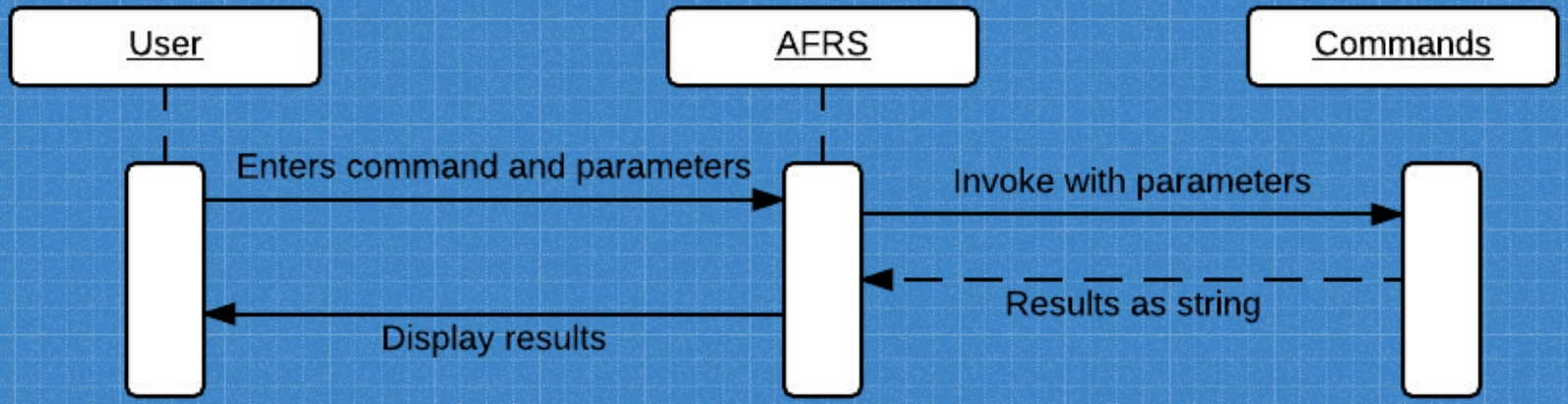
Design

- Command pattern
  - HashMap
- String splitting
- Simple single-method class

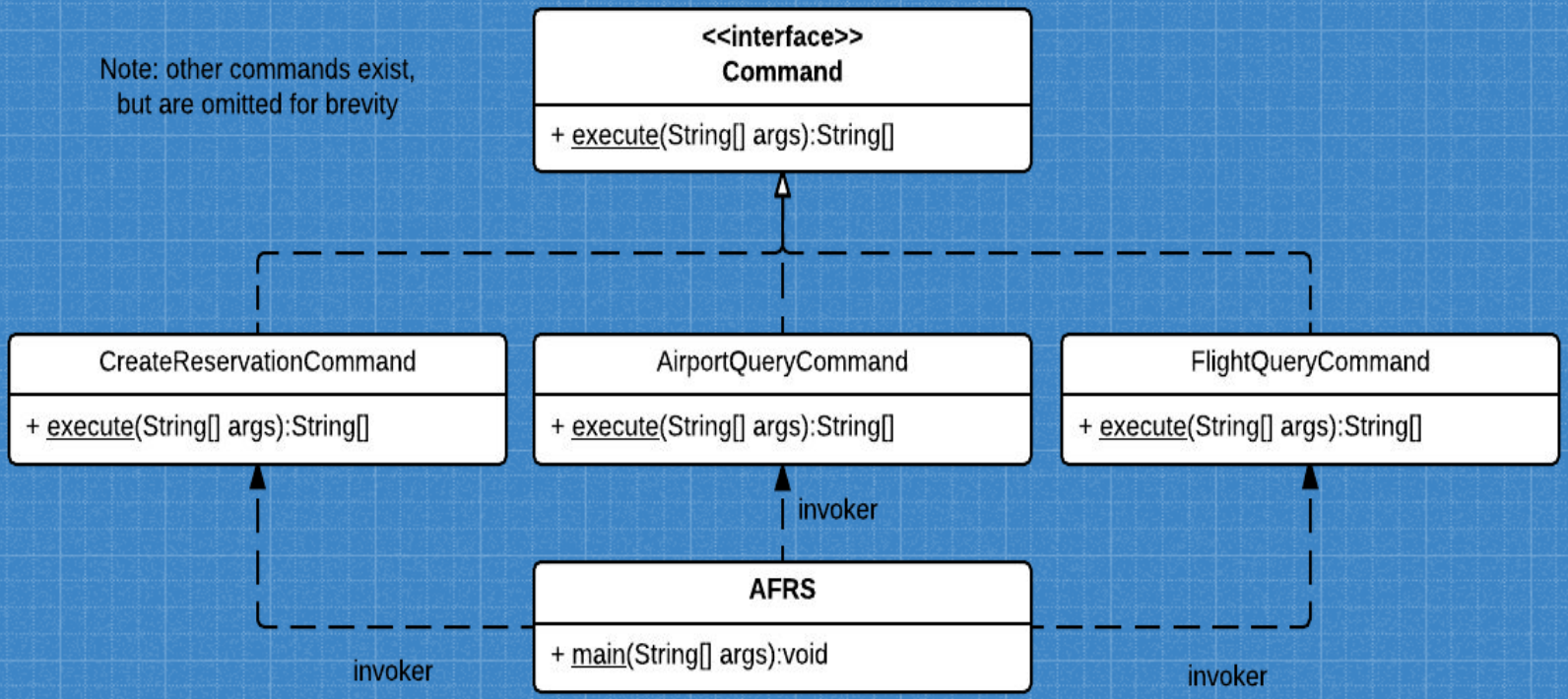
GoF Card

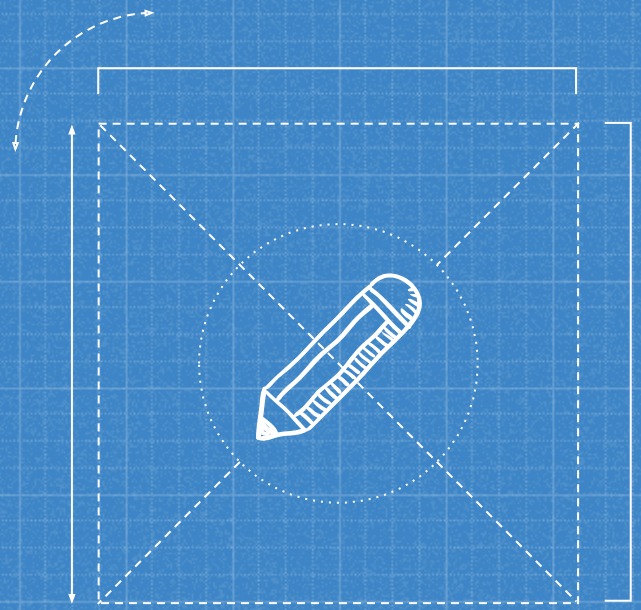
Subsystem name: UI		**Pattern**: Command
Participants		
Class	**Role**	**Participant's contributions**
Command	Command interface	Defines the execute method for the commands
AFRS	Client	Creates the commands
AFRS	Invoker	Invokes the commands
FlightQueryCommand	Concrete command	Implements the logic to query for flights
CreateReservationCommand	Concrete command	Implements the logic to create a reservation
...		
Reservation	Reciever	Handles the creation, deletion, and retrieval of reservations
...		
Deviations	The AFRS class is both the client and the invoker. Usually they're separate	
Requirements fulfilled	Requirement 1	

UI Sequence


User
AFRS
Commands
Enters command and parameters
Invoke with parameters
Results as string
Display results

UI UML


Note: other commands exist, but are omitted for brevity
<<interface>>
Command
+ execute(String[] args):String[]
CreateReservationCommand
+ execute(String[] args):String[]
AirportQueryCommand
+ execute(String[] args):String[]
FlightQueryCommand
+ execute(String[] args):String[]
invoker
AFRS
+ main(String[] args):void
invoker
invoker



Airport Information Subsystem

Requirements

- Shall allow a client to query for information about an airport by inputting a specific airport code such as:
  - Airport Name
  - Weather
  - Temperature
  - Delay Time
- If the client enters a null or invalid airport code, then the system will display an error message.
- Output:
  - (airport, [airportName], [airportWeather], [airportTemp], [airportCurrent Delay]).

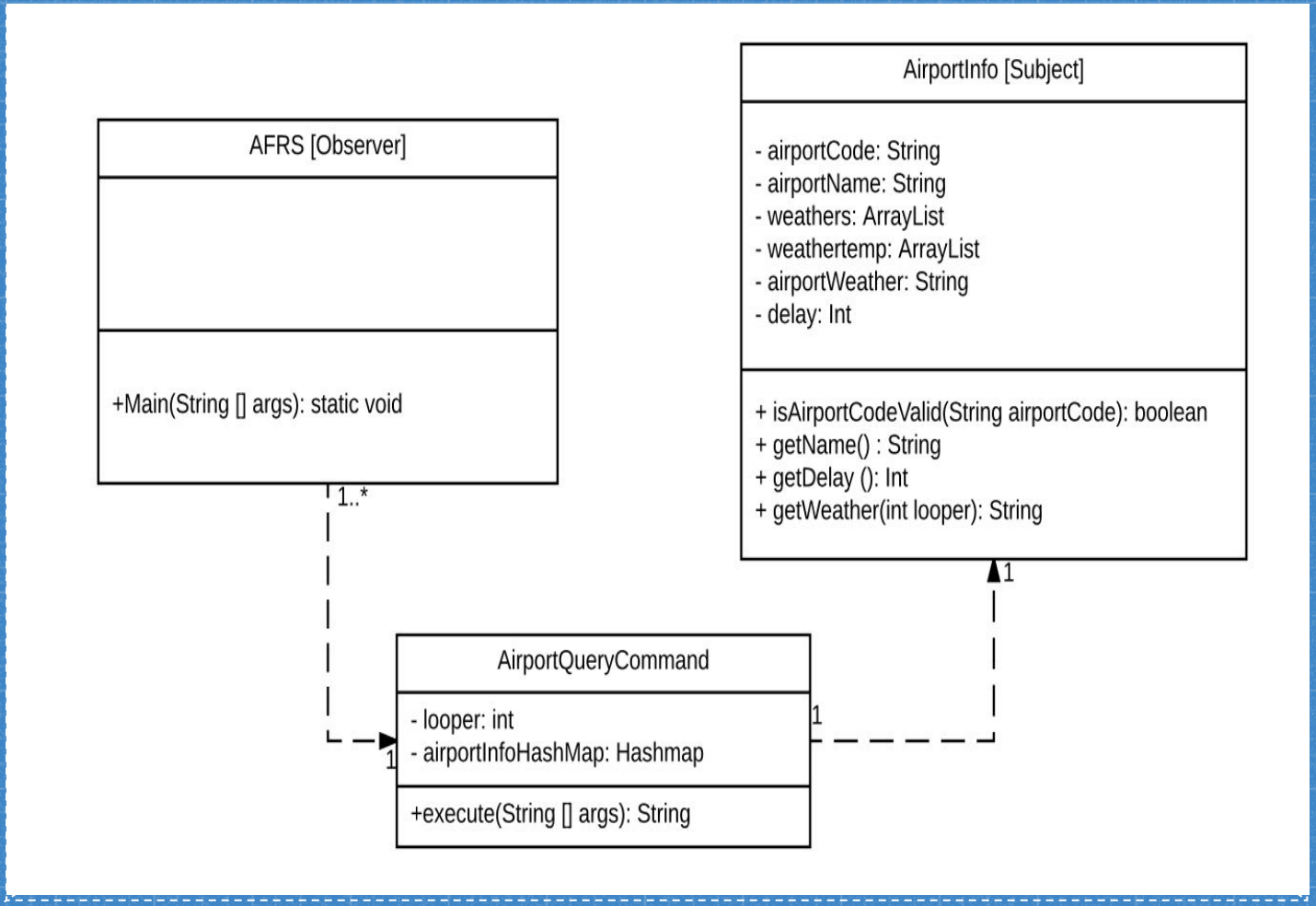
Description

- AFRS sends argument (airport code) to a concrete command class (AirportQueryCommand)
- AirportQueryCommand then sends argument to AirportInfo and calls GetAirportInformation
- This information is then stored in a Hashmap
  - Weather and Temperature are stored in another arraylist so that each set can be iterated through
- The Information is then retrieved and but created into a string

Gof Card

Subsystem Name: Airport Information		**GoF pattern:** Observer Pattern
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
AirportInfo	Subject	Once the AirportInfo has been queried for in the AFRS user interface class, then AirportInfo will send (“pushes”) the data payload of the new weather and temperature (once it has been updated) and notifies the AFRS (user interface)
N/A	Observer	We did not implement an observer interface since there is only one observer
AFRS	Concrete Observer	The client using the AFRS user interface, will make a query in this class about airport information and will pull from the subject (AirportInfo) the information that is needed (airport name, weather, temp, current delay time). This observer, will want to be notified of the weather and temperature, after it queries and it has been changed.
Deviations from the standard pattern: There is a deviation in this pattern. Since there is only one observer (AFRS), we did not find it necessary to implement an abstract observer or interface for this pattern. The observer also gets notified only when it queries for airport information.		
Requirements being covered: Requirement 1a, 1b, and 1c, Requirement 3i and 3ii		

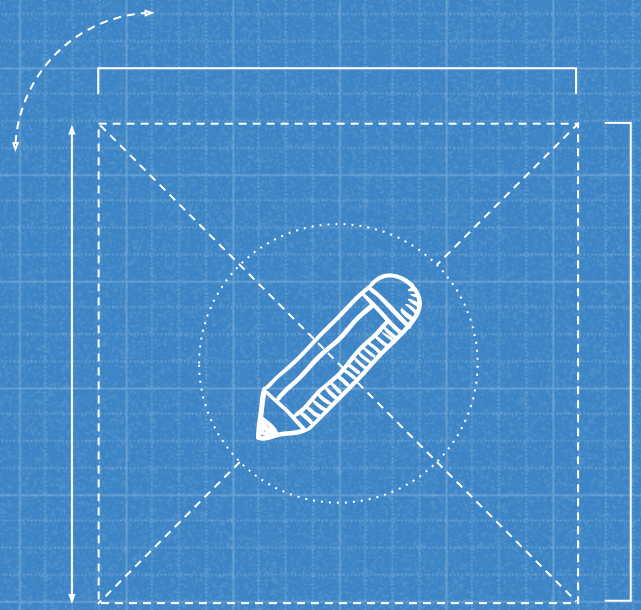
UML Diagram


AFRS [Observer]
+Main(String [] args): static void
AirportInfo [Subject]
- airportCode: String
- airportName: String
- weathers: ArrayList
- weathertemp: ArrayList
- airportWeather: String
- delay: Int
+ isAirportCodeValid(String airportCode): boolean
+ getName() : String
+ getDelay (): Int
+ getWeather(int looper): String
AirportQueryCommand
- looper: int
- airportInfoHashMap: Hashmap
+execute(String [] args): String
1..*
1
1
1

Design Rationale

- ***Observer Pattern***:
  - Difficult to find a pattern to specifically use for this subsystem
  - Rationale for choosing observer pattern was that the weather and temperature is subject to change since there are many alternatives,
  - The user interface would be the observer, and be notified, anytime that the weather changes, so that if an airport is queried for, the proper weather and temperature is shown

- ***AirportInfo class:***
  - What type of data structure should hold the airport information so that it can easily be accessible by other classes
    - Initially, decided to create an arraylist to hold a airport code (String), airport name (String), a list of weather and temperatures (Arraylist), and delay times (int)
    - Later decided a hashmap would be better suited for this type of implementation
      - However, we ran out of time to implement in this release



Flight Query Subsystem

Requirements

- Shall allow a client to query flight information. The system then returns a list of all possible itineraries that are sorted by arrival, departure, or airfare
  -
- If the client enters a null or invalid airport code, then the system will display an error message. If no itineraries exist it will return none.
- Output:
  - info, [number of results] <\n>
  - [itinerary id],[airfare],[connections],[fligthinfo]

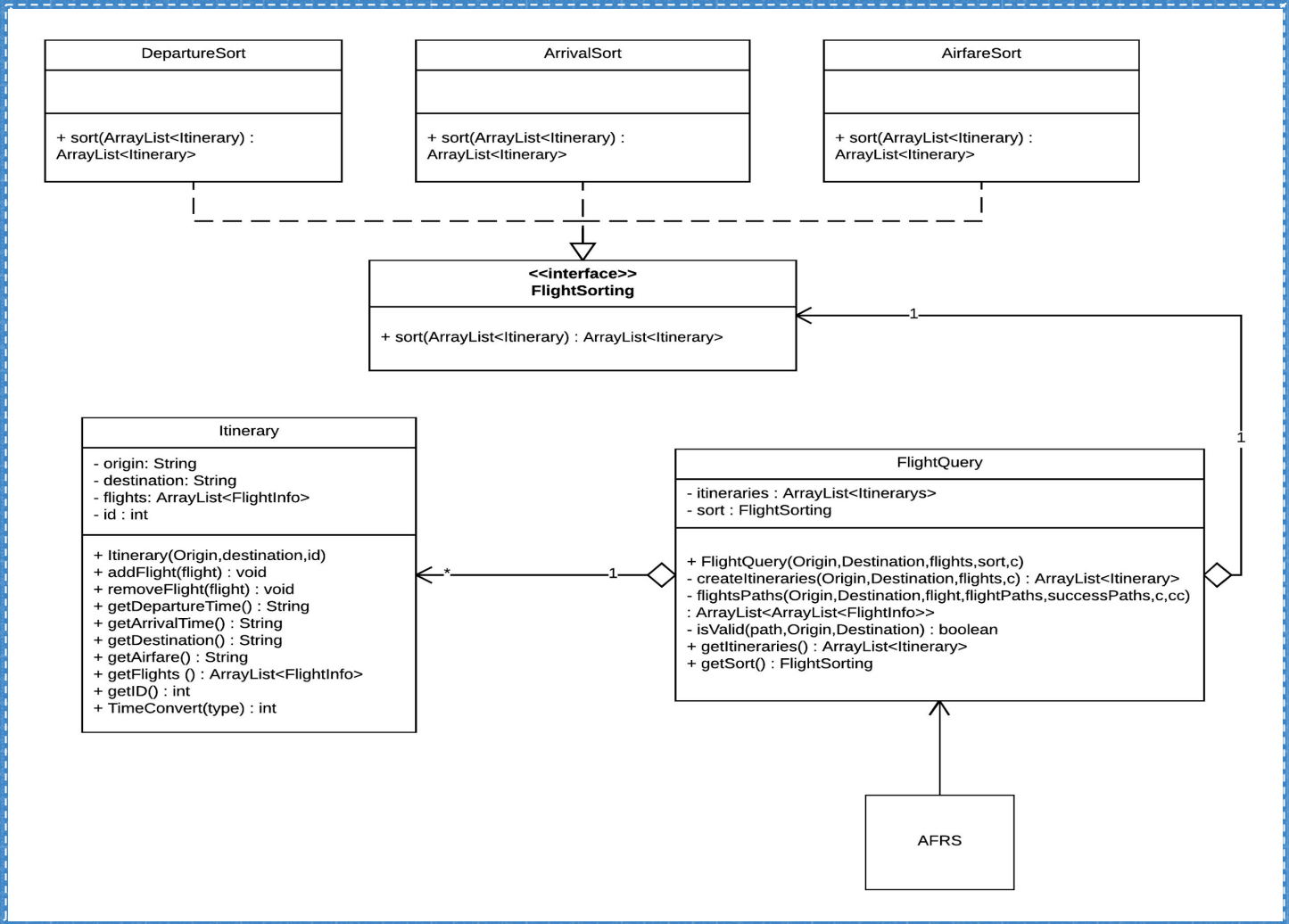
Description

- AFRS sends arguments to a concrete command class (FlightQueryCommand)
- FlightQuery is called and begins finding all possible routes from origin
- The routes found from the origin to the destination are then time validated by the isValid method
- Every valid flight path is then created into an Itinerary.
- The sub-system also uses the Strategy pattern for sorting the returned itineraries.

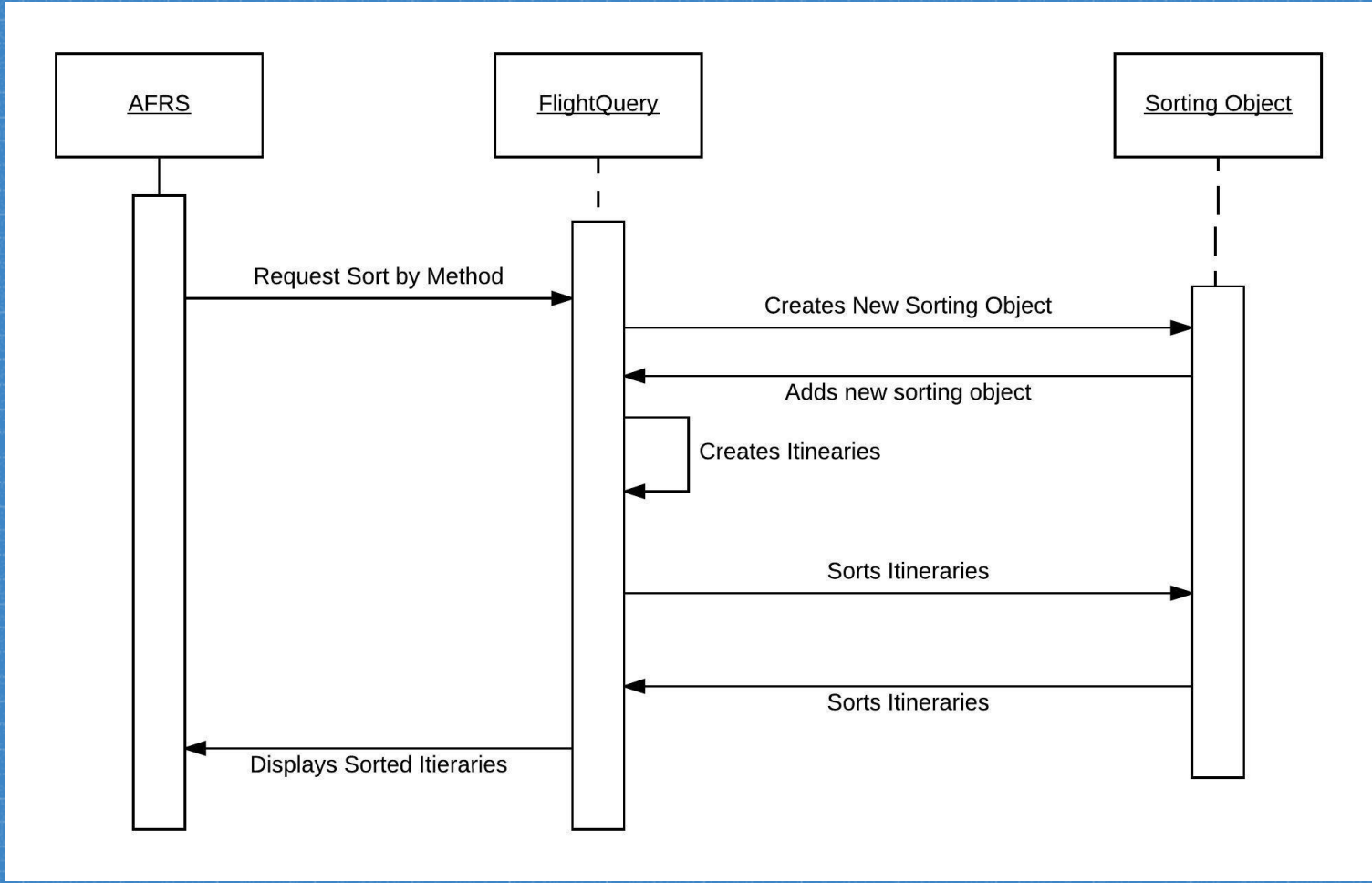
Gof Card

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

UML Diagram


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

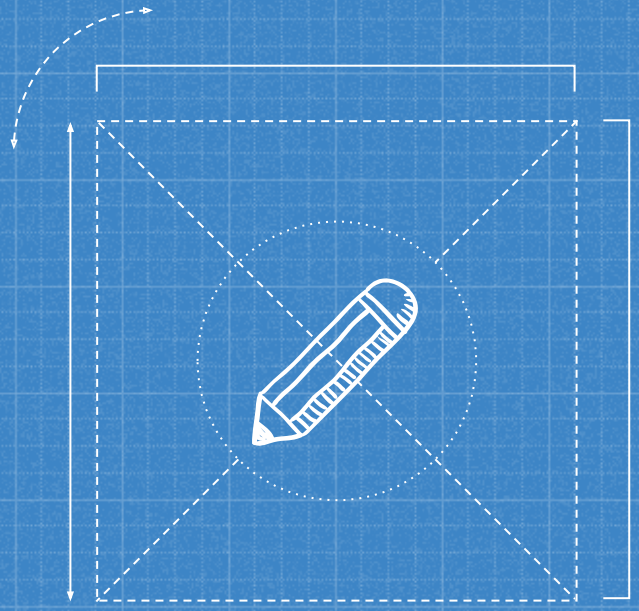
Sequence Diagram



Design Rationale

- ***Strategy Pattern:***
  - *The different sorting algorithms are perfect for the strategy pattern. Each sorting type changes the behavior of the system and since the sorting algorithm is chosen by the user, we have a strategy pattern on our hands.*

- ***Hashmap over ArrayList:***
  - *Originally I used an arraylist to store all the flights. The system had performance problems when finding all possible paths out of the airports. I originally thought this was do to having to iterate over an arraylist over and over again, so I switched to a hashmap where the keys were the airport codes and the values were the flights.*



Itinerary Subsystem

Requirements

- This subsystem deals with the creation of Itineraries which store:
  - Destination
  - Origin
  - Flights

- An Itinerary must also be able to calculate the total airfare of the flights
- The flights in an Itinerary must be valid. Flight one must arrive before flight two leaves while accounting for connection time and airport delays.

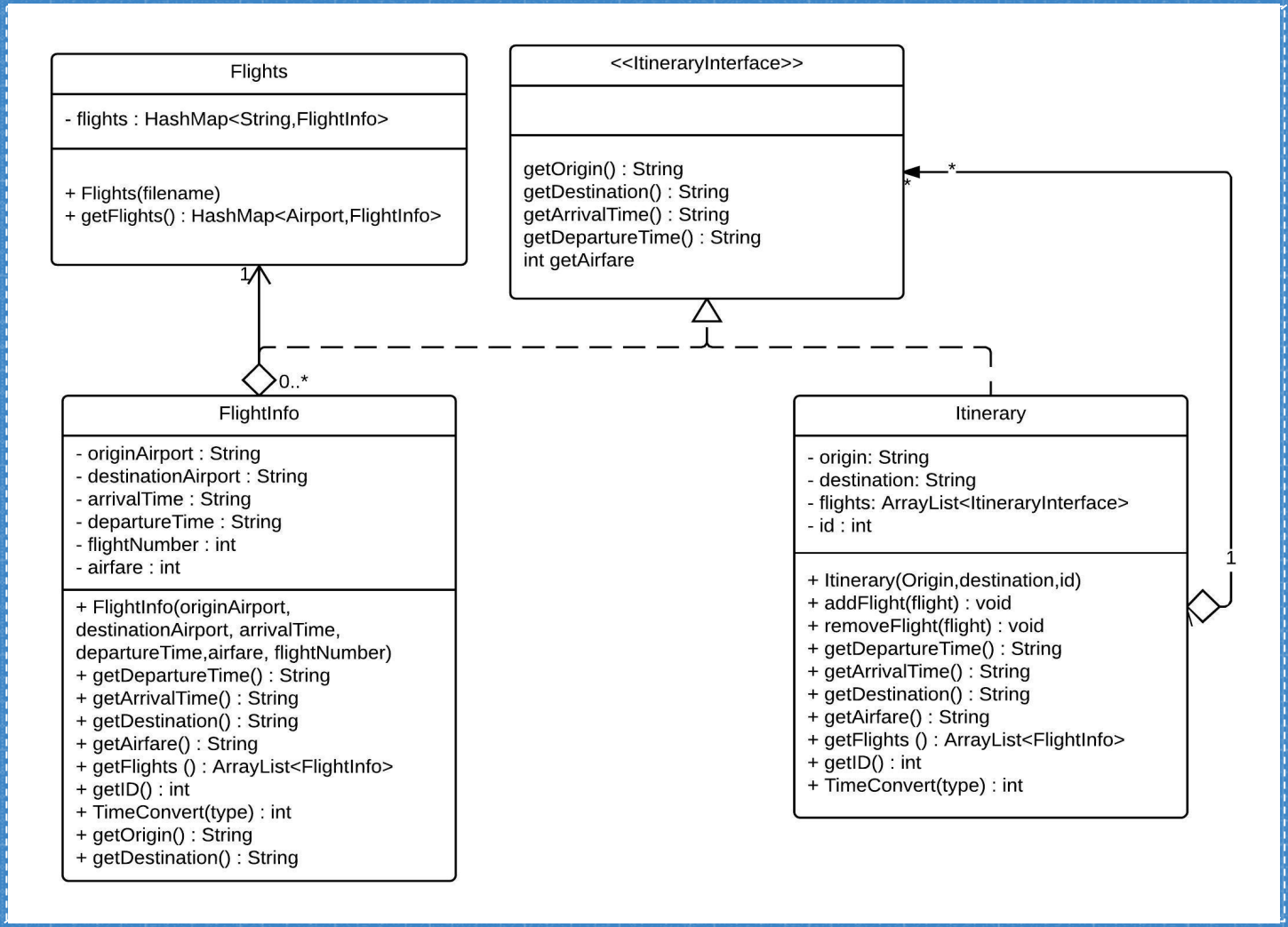
Description

- The Itinerary is created when a flight is queried and the path is validated during the query.
- The Itinerary Subsystem uses the Composite pattern where the Itinerary is the composite and the Flights are the leaf.

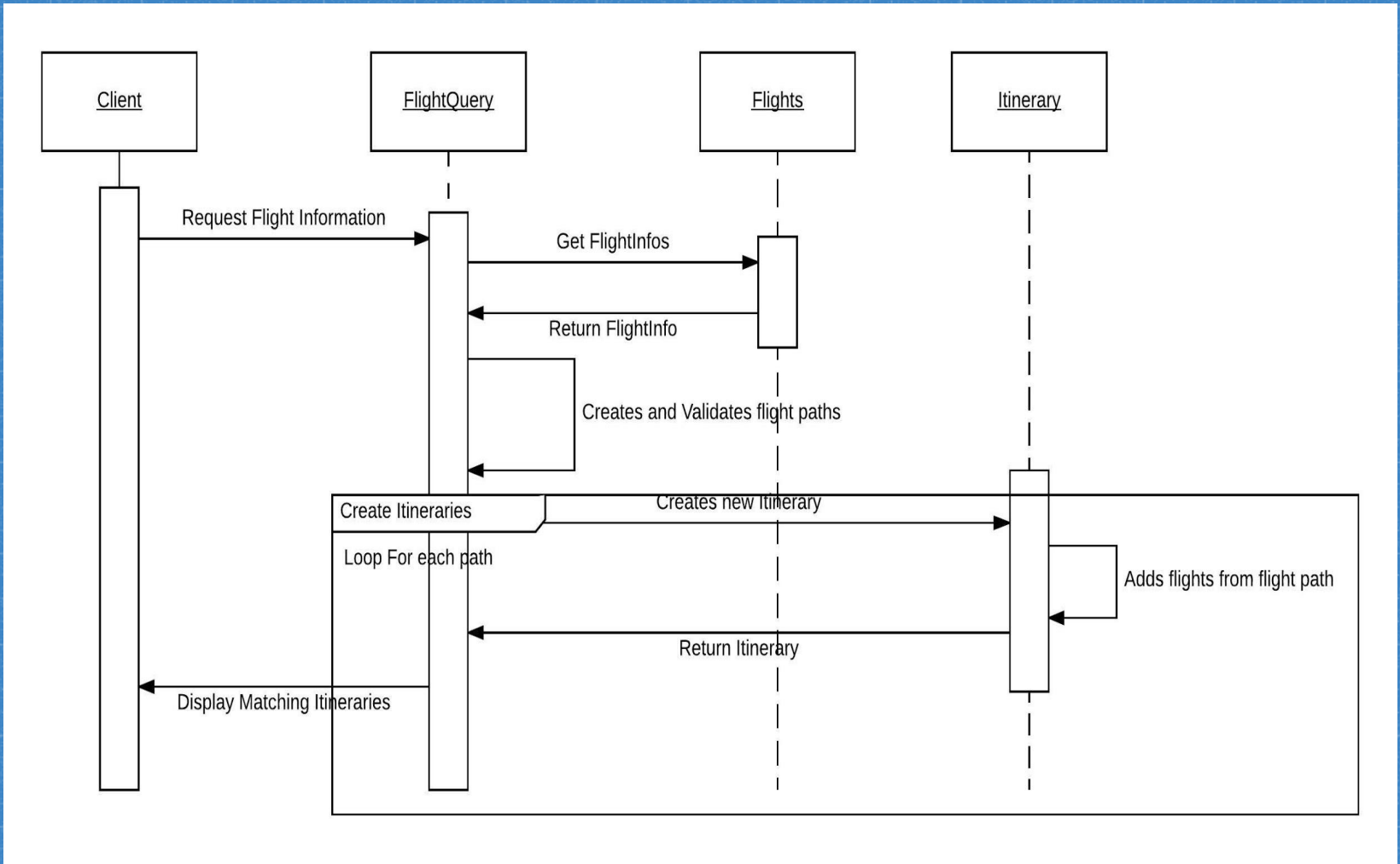
Gof Card

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

UML Diagram


Flights
- flights : HashMap<String,FlightInfo>
+ Flights(filename)
+ getFlights() : HashMap<Airport,FlightInfo>
<<ItineraryInterface>>
getOrigin() : String
getDestination() : String
getArrivalTime() : String
getDepartureTime() : String
int getAirfare
1
0..*
*
*
1
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

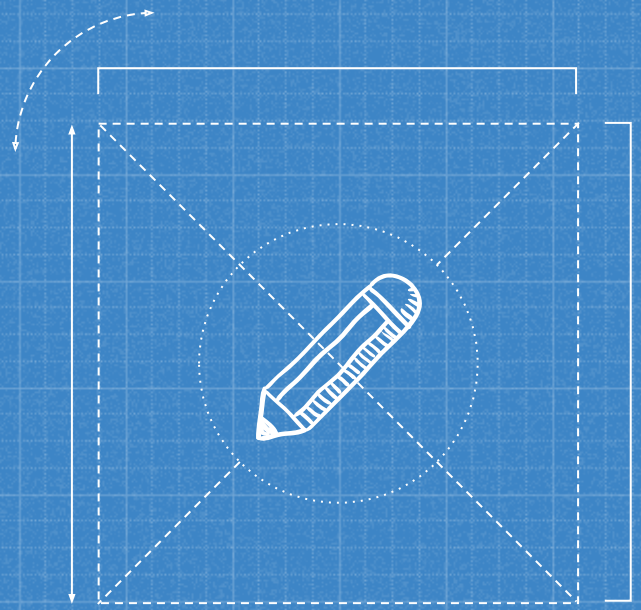
Sequence Diagram



Design Rationale

- ***Composite pattern:***
  - *The composite pattern was chosen because it allowed us to create different length itineraries. Since an itinerary can hold up to three flights, the composite pattern allowed us to create a fluid class that could hold from one flight to three flights.*

- ***The Flight Class:***
  - ***A small discussion was had over how to hold flight information in the system. It was decided that we needed to create all flightInfo objects on start up and store them. This is what the flight class does. It creates them on start up and saves them in a state for call later in the life of the system.***



Reservation Subsystem

Requirements

- Create
  - Per-passenger
  - Based on itinerary from flight query
  - One per passenger and destination/origin pair
- Retrieve
  - Passenger, optional origin/destination
- Delete
  - Passenger, origin, destination
- Persist

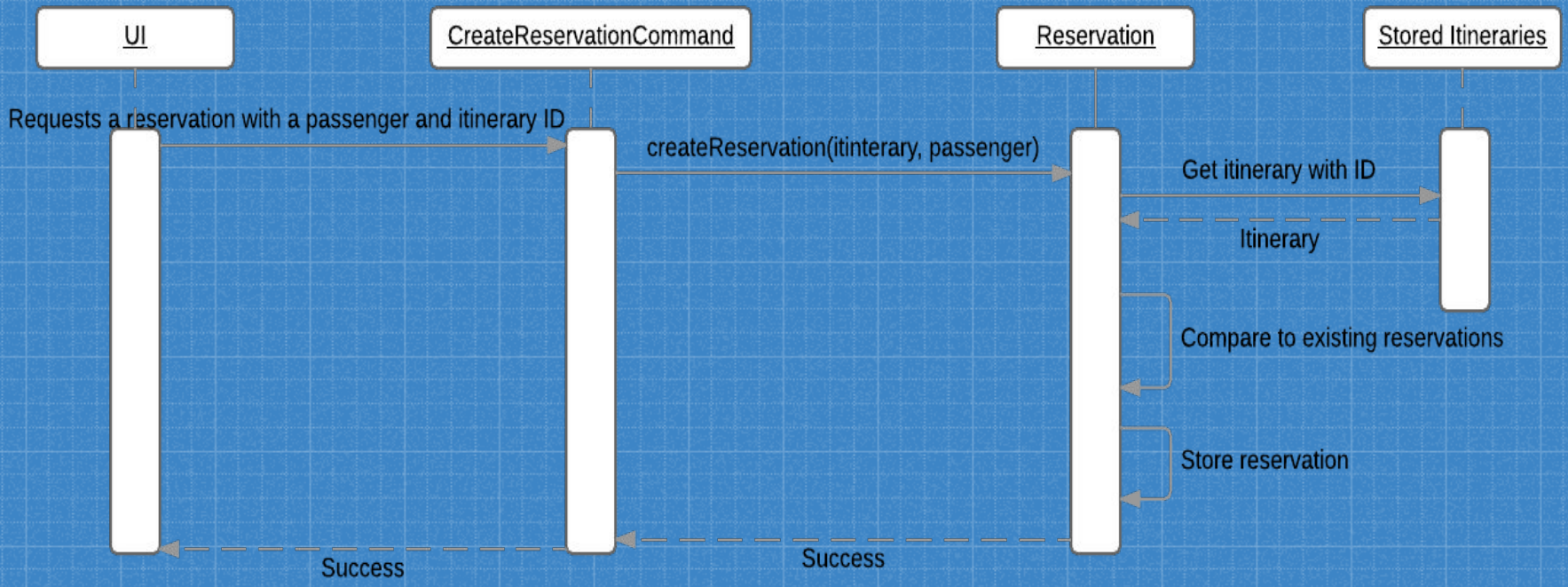
Design

- Incoming commands

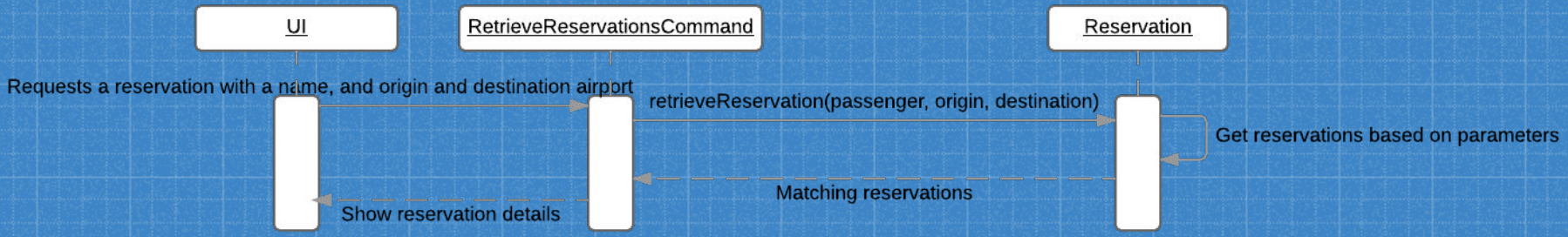
GoF card

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

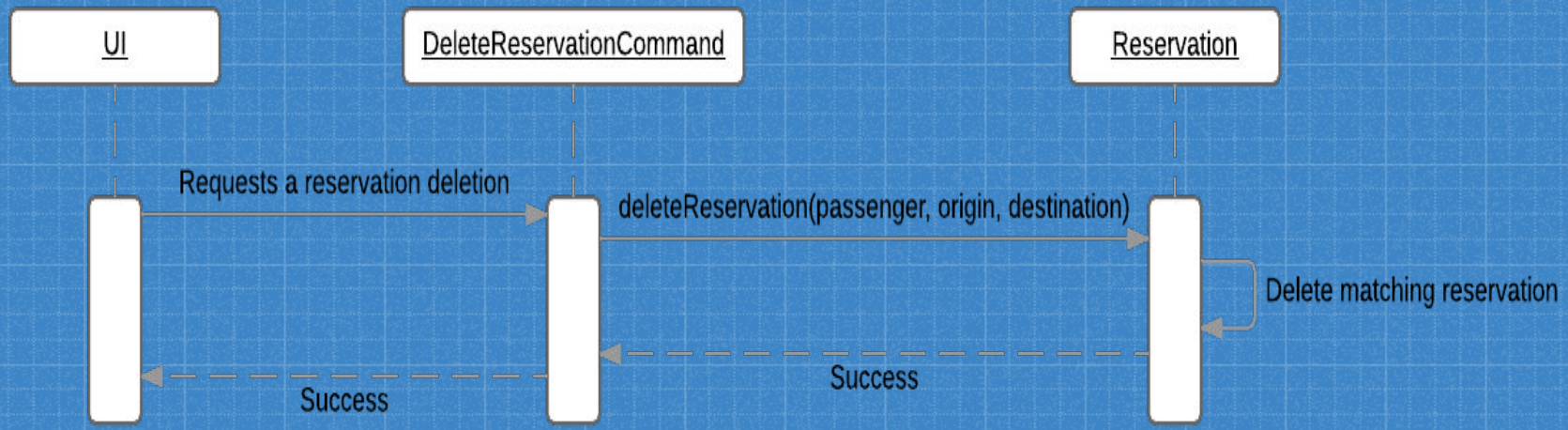
Create reservation sequence



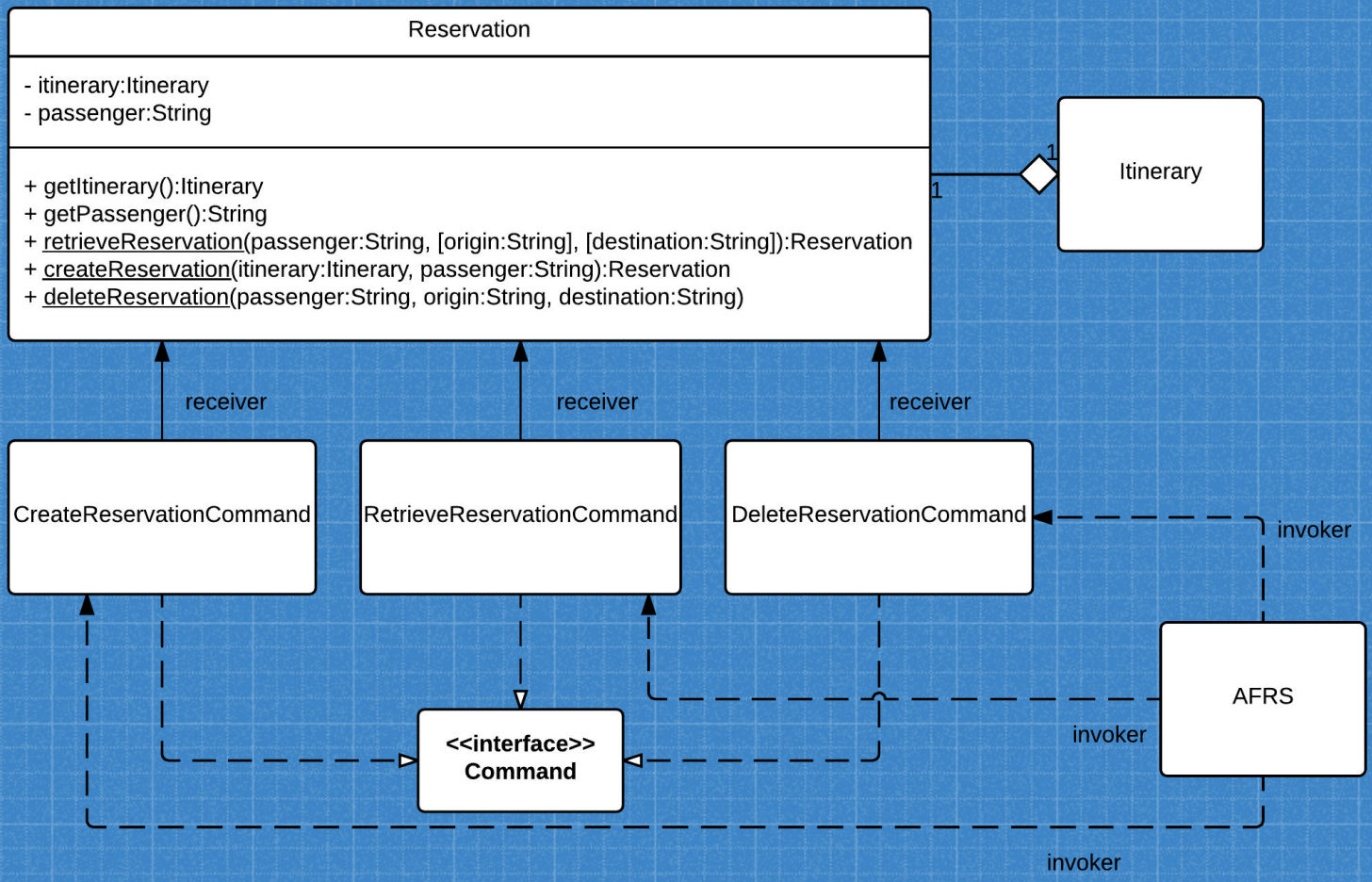
Retrieve reservation sequence



Delete reservation sequence



Reservation UML



4

Strengths and Weaknesses of our Design

Strengths and Weaknesses

- Strengths:
  - **Open-Close Principle**: Adding sorting methods and commands
  - **Separation of Concerns**: little overlap in functionality, each class and subsystem was responsible only for the data that it was concerned with
  - **Don’t repeat yourself (DRY)**: Eliminated a lot of lava flow and commented out code
- Weaknesses:
  - Runs a little Slow for some larger airports (LAX) with larger connections (2)
  - Reservations are not persisted

Thanks!

ANY QUESTIONS?