```
/*
 * File:   Airport.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import model.data_store.AirportStore;
import model.data_store.FAAAirport;
import model.data_store.ServerStore;

public class Airport implements Command {


    ServerStore instance = ServerStore.getInstance();
    static ServerStore.Server currServer;

    private String UNKNOWN_AIRPORT = "error,unknown airport";

    /**
     * Args: String airport-name, String (weather,temp), int delay
     */
    private String RESPONSE_FORMAT = "airport,%s,%s,%d";

    private String request;

    private String airportCode;

    private int clientID;

    /**
     * Create the Airport command, which gets the information about an Airport.
     *
     * @param request the request parameters
     *                for Airport: airport
     *                where airport is the three-letter code of the airport
     * @param clientID the identification number of the client who made the request
     */
    public Airport(String request, int clientID) {
        this.request = request;
        this.clientID = clientID;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        String parseResult = parseRequest();

        if (parseResult != null) {
            return parseResult;
        }
        getServer(clientID);
        if (currServer == ServerStore.Server.local) {
            AirportStore airportStore = AirportStore.getInstance();
            String returnString = airportStore.getAirportName(airportCode) + "," +
airportStore.getAirportWeather(clientID, airportCode) + "," +
airportStore.getAirportDelay(airportCode);
            return returnString;
        }
        if (currServer == ServerStore.Server.faa) {
            FAAAirport instance = FAAAirport.getInstance();
            String returnString = instance.getAirportName(airportCode) + "," +
instance.getAirportWeather(clientID, airportCode) + ","
                    + instance.getAirportDelay(airportCode);
            return returnString;

        }
        return "";
    }

    /**
     * Parse the request string and set the airport field with the relevant airport
object.
     *
     * @return null if parse was successful, or an error message to be sent back to
the client if unknown airport
     */
    private String parseRequest() {
        if (request == null || request.split(",").length > 1) {
            return INVALID_NUMBER_PARAMS;
        }

        AirportStore airportStore = AirportStore.getInstance();
        airportCode = request;

        if (!airportStore.isAirport(airportCode)) {
            return UNKNOWN_AIRPORT;
        }

        return null;
    }
    private void getServer(int id) {
       if(instance.getServer(id) == null) {

       } else {
           currServer = instance.getServer(id);
       }
    }
}

/*
 * File:   Command.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

public interface Command {

    String INVALID_NUMBER_PARAMS = "error,invalid number of parameters";

    String UNKNOWN_ORIG = "error,unknown origin";

    String UNKNOWN_DEST = "error,unknown destination";

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    String execute();

}
/*
 * File:   Delete.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import model.components.Flight;
import model.components.Itinerary;
import model.components.TripComponent;
import model.data_store.ReservationStore;

import java.text.SimpleDateFormat;
import java.util.List;
```

```
public class Delete implements UndoableCommand {

    private String request;

    private String passenger;

    private String origin;

    private String destination;

    private boolean runSuccess;

    private Itinerary deletedItin;

    private int clientID;

    private static String DUPLICATE = "error,duplicate reservation";

    static String SUCCESS = "delete,successful";

    static String RESERVATION_NOT_FOUND = "error,reservation not found";

    private static String REDO_RESP_FORMAT = "redo,delete,%s,%s";

    private static String UNDO_RESP_FORMAT = "undo,delete,%s,%s";

    private static String FLIGHT_FORMAT = ",%d,%s,%s,%s,%s";

    private static String DATE_FORMAT = "h:mma";

    private static SimpleDateFormat DATE_FORMATTER = new
SimpleDateFormat(DATE_FORMAT);

    /**
     * Create the delete command, which deletes a reservation for a given passenger.
     *
     * @param request the request parameters
     *                for Delete: passenger,origin,destination
     *                where passenger is the name of the passenger holding the
reservation,
     *                origin is the three-letter code for the reservation's origin
airport,
     *                destination is the three-letter code for the reservation's
destination airport
     * @param clientID the identification number of the client who made the request
     */
    public Delete(String request, int clientID) {
        this.request = request;
        this.clientID = clientID;
        this.runSuccess = false;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        runSuccess = false;

        String parseResult = parseRequest();
        if (parseResult != null) {
            return parseResult;
        }

        ReservationStore reservationStore = ReservationStore.getInstance();

        deletedItin = reservationStore.delete(passenger, origin, destination);

        if (deletedItin == null) {
            return RESERVATION_NOT_FOUND;
        }

        runSuccess = true;
        return Delete.SUCCESS;
    }

    /**
     * Parse the request string and set the passenger, origin, and destination 
fields.
     *
     * @return null if parse was successful, or an error message to be sent back to 
the client
     */
    private String parseRequest() {
        String[] requestArr;
        if (request == null || !((requestArr = request.split(",")).length == 3)) {
            return INVALID_NUMBER_PARAMS;
        }

        passenger = requestArr[0];
        origin = requestArr[1];
        destination = requestArr[2];
        return null;
    }

    /**
     * Undo this command.
     *
     * @return output to be directed to the client
     */
    @Override
    public String undo() {
        runSuccess = false;

        ReservationStore reservationStore = ReservationStore.getInstance();

        ReservationStore.Result result = reservationStore.reserve(passenger,
deletedItin);

        switch (result) {
            case ALREADY_RESERVED:
                return DUPLICATE;
        }

        runSuccess = true;

        StringBuilder itineraryString = new StringBuilder();

        List<Flight> flights = deletedItin.getFlights();

itineraryString.append(deletedItin.getAirfare()).append(",").append(flights.size());

        for (TripComponent flight : flights) {
            itineraryString.append(String.format(FLIGHT_FORMAT, flight.getId(),
flight.getOrigin().getName(),
                    DATE_FORMATTER.format(flight.getArrival()),
                    flight.getDestination().getName(),
DATE_FORMATTER.format(flight.getDeparture())));
        }

        return String.format(UNDO_RESP_FORMAT, passenger,
itineraryString.toString());
    }

    /**
     * Redo this command.
     *
     * @return output to be directed to the client
     */
    @Override
    public String redo() {
        runSuccess = false;

        ReservationStore reservationStore = ReservationStore.getInstance();

        ReservationStore.Result result = reservationStore.delete(passenger,
deletedItin);

        if (result.equals(ReservationStore.Result.NOT_FOUND)) {
            return RESERVATION_NOT_FOUND;
        }

        runSuccess = true;

        StringBuilder itineraryString = new StringBuilder();

        List<Flight> flights = deletedItin.getFlights();

itineraryString.append(deletedItin.getAirfare()).append(",").append(flights.size());

        for (TripComponent flight : flights) {
            itineraryString.append(String.format(FLIGHT_FORMAT, flight.getId(),
flight.getOrigin().getName(),
                    DATE_FORMATTER.format(flight.getArrival()),
                    flight.getDestination().getName(),
DATE_FORMATTER.format(flight.getDeparture())));
        }

        return String.format(REDO_RESP_FORMAT, passenger,
itineraryString.toString());
    }

    /**
     * Did this command run without error on last execution.
     *
     * @return true if the command ran without error, false if any error occurred
     */
    @Override
    public boolean wasSuccessful() {
        return runSuccess;
    }

    /**
     * Get the ID of the client which made this request.
     *
     * @return client ID number
     */
    @Override
    public int getClientID() {
        return clientID;
    }

}
/*
 * File:   Info.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import model.components.*;
import model.data_store.AirportStore;
import model.data_store.ItineraryStore;

import java.text.SimpleDateFormat;
import java.util.List;

public class Info implements Command {

    private static String DATE_FORMAT = "h:mma";

    private static SimpleDateFormat DATE_FORMATTER = new
SimpleDateFormat(DATE_FORMAT);

    private static String INVALID_NUM_CONNECT = "error,invalid connection limit";

    private static String INVALID_SORT_ORDER = "error,invalid sort order";

    private static String FLIGHT_FORMAT = ",%d,%s,%s,%s,%s";

    private static int MAX_CONNECTIONS = 2;

    private enum SORT_ORDER {
        departure,
        arrival,
        airfare
    }

    private String request;

    private String origin;

    private String destination;

    private int connections = -1;
    private int clientID;

    private SORT_ORDER sortOrder = SORT_ORDER.departure;

    /**
     * Create the Info command, which acquires the itineraries which fit the
client's parameters.
     *
     * @param request the request parameters
     *                for Info: origin,destination[,connections[,sort-order]]
     *                where origin is the three-letter code for the origin airport,
     *                destination is the three-letter code for the destination
airport,
     *                connections is the maximum number of connections in the
itinerary (0-2),
     *                sort-order is the method of sorting the itineraries
(departure, arrival, airfare)
     * @param clientID the identification number of the client who made the request
     */
    public Info(String request, int clientID) {
        this.request = request;
        this.clientID = clientID;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        String parseResult = parseRequest();
        if (parseResult != null) {
            return parseResult;
        }

        ItineraryStore itineraryStore = ItineraryStore.getInstance();
        List<Itinerary> itineraries;
        if (connections == -1) {
            itineraries = itineraryStore.getItineraries(origin, destination);
        } else {
            itineraries = itineraryStore.getItineraries(origin, destination,
connections);
        }

        switch (sortOrder) {
            case departure:
                itineraries.sort(new DepartureComparator());
                break;
            case arrival:
                itineraries.sort(new ArrivalComparator());
                break;
            case airfare:
                itineraries.sort(new AirfareComparator());
                break;
        }

        InfoQueryStore infoQueryStore = InfoQueryStore.getInstance();
        infoQueryStore.setItineraries(clientID, itineraries);

        StringBuilder output = new StringBuilder("info," + itineraries.size());

        int itineraryID = 1;
        for (Itinerary itinerary : itineraries) {
            output.append("\n").append(itineraryID).append(",");
            List<Flight> flights = itinerary.getFlights();

output.append(itinerary.getAirfare()).append(",").append(flights.size());

            for (TripComponent flight : flights) {
                output.append(String.format(FLIGHT_FORMAT, flight.getId(),
flight.getOrigin().getName(),
                        DATE_FORMATTER.format(flight.getArrival()),
                        flight.getDestination().getName(),
DATE_FORMATTER.format(flight.getDeparture())));
            }

            itineraryID++;
        }

        return output.toString();
    }

    /**
     * Parse the request string and set the fields with given values.
     *
     * @return null if parse was successful, or an error message to be sent back to
the client
     */
    private String parseRequest() {
        String[] requestArr;
        if (request == null || ((requestArr = request.split(",")).length > 4) ||
requestArr.length < 2) {
            return INVALID_NUMBER_PARAMS;
        }
        AirportStore airportStore = AirportStore.getInstance();

        origin = requestArr[0];
        destination = requestArr[1];

        if (!airportStore.isAirport(origin)) {
            return UNKNOWN_ORIG;
        }
        if (!airportStore.isAirport(destination)) {
            return UNKNOWN_DEST;
        }

        try {  // parse optional connection limit
            if (!requestArr[2].equals("")) {
                connections = Integer.parseInt(requestArr[2]);

                if (connections > MAX_CONNECTIONS || connections < 0) {
                    return INVALID_NUM_CONNECT;
                }
            }
        } catch (IndexOutOfBoundsException ignored) {  // no connection limit 
specified, use default
        } catch (NumberFormatException e) {  // failure to parse string into int => 
invalid connection limit
            return INVALID_NUM_CONNECT;
        }

        try {  // parse optional sort order
            if (!requestArr[3].equals("")) {
                sortOrder = SORT_ORDER.valueOf(requestArr[3].toLowerCase());
            }
        } catch (IndexOutOfBoundsException e) {  // no sort order specified, use 
default
        } catch (IllegalArgumentException e) {  // failure to find sort order enum 
value
            return INVALID_SORT_ORDER;
        }

        return null;
    }

}
/*
 * File:   InfoQueryStore.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import control.request_response.ClientIdentifierObserver;
import control.request_response.ClientIdentifierSubject;
import model.components.Itinerary;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Stores the most recent set of itineraries given from an Info command. This set is
used by the Reserve command.
 * Reservations are made based on an itinerary ID given during output. The ID is the
position within the set of
 * itineraries starting from 1.
 *
 * This is a singleton.
 */
public class InfoQueryStore implements ClientIdentifierObserver {

    private static InfoQueryStore instance;

    private Map<Integer, List<Itinerary>> latestGivenItins;

    private InfoQueryStore(ClientIdentifierSubject clientIDManager) {
        latestGivenItins = new HashMap<>();
        instance = this;
        clientIDManager.attachObserver(this);  // observe the ID manager
    }

    /**
     * Get the singleton instance of the InfoQueryStore. T
     *
     * @return instance of the store
     */
    public static InfoQueryStore getInstance(ClientIdentifierSubject
clientIDManager) {
        if (instance == null) {
            return new InfoQueryStore(clientIDManager);
        }
        return instance;
    }

    /**
     * Get the singleton instance of the InfoQueryStore.
     *
     * @return instance of the store
     */
    public static InfoQueryStore getInstance() {
        if (instance == null) {
            throw new RuntimeException("InfoQueryStore was not properly instantiated 
before a call to getInstance");
        }
        return instance;
    }

    /**
     * Set the most recent list of Itineraries given to a client in the Info 
command.
     *
     * @param clientID the client identification number to set given itineraries for
     * @param itineraries list of Itineraries given to client
     */
    void setItineraries(int clientID, List<Itinerary> itineraries) {
        latestGivenItins.put(clientID, itineraries);
    }

    /**
     * Get an itinerary from its position in the output.
     *
     * @param clientID the client identification number to get the itinerary for
     * @param itineraryID ID of the itinerary to get (position in output)
     * @return Itinerary object for that ID, or null if the ID is invalid
     */
    Itinerary getItinerary(int clientID, int itineraryID) {
        if (isValidID(clientID, itineraryID)) {
            return latestGivenItins.get(clientID).get(itineraryID - 1);
        }
        return null;
    }

    /**
     * Check if the ID for an itinerary is valid based on the last given list of 
Itineraries. (checks if in bounds)
     *
     * @param clientID the client identification number to check the itinerary ID 
for
     * @param itineraryID ID of the itinerary (position in output)
     * @return true if ID is valid, false otherwise
     */
    boolean isValidID(int clientID, int itineraryID) {
        return latestGivenItins.containsKey(clientID) &&
                (itineraryID >= 1) && (itineraryID <= 
latestGivenItins.get(clientID).size());
    }

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    @Override
    public void notifyNewClient(int clientID) {
        latestGivenItins.put(clientID, new ArrayList<>());
    }

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    @Override
    public void notifyClientDisconnect(int clientID) {
        latestGivenItins.remove(clientID);
    }
}
package control.request_response.commands;

/**
 * The redo command allows UndoManager to know to redo a command.
 */
public class Redo implements Command {

    private boolean validRedo;

    private int clientID;

    /**
     * Create the redo command.
     *
     * @param request the request parameters, null if no params
     * @param clientID IDF number of client which made request
     */
    public Redo(String request, int clientID) {
        this.clientID = clientID;
        this.validRedo = request == null;  // redo command should have no request
parameters
    }

    /**
     * Was this redo request valid?
     *
     * @return true if the redo request was valid, false otherwise
     */
    public boolean isValid() {
        return validRedo;
    }

    /**
     * Get the ID of the client which made the redo request.
     *
     * @return client ID number
     */
    public int getClientID() {
        return clientID;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        if (!validRedo) {
            return INVALID_NUMBER_PARAMS;
        }
        return null;  // result of re-done command will be response, so this return
is ignored
    }
}
/*
 * File:   Reserve.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import model.components.Flight;
import model.components.Itinerary;
import model.components.TripComponent;
import model.data_store.ReservationStore;

import java.text.SimpleDateFormat;
import java.util.List;

public class Reserve implements Command, UndoableCommand {
    private static String INVALID_ID = "error,invalid id";

    private static String DUPLICATE = "error,duplicate reservation";

    private static String RESERVATION_NOT_FOUND = "error,reservation not found";

    private static String SUCCESS = "reserve,successful";

    private static String FLIGHT_FORMAT = ",%d,%s,%s,%s,%s";

    private static String UNDO_RESP_FORMAT = "undo,reserve,%s,%s";

    private static String REDO_RESP_FORMAT = "redo,reserve,%s,%s";

    private static String DATE_FORMAT = "h:mma";

    private static SimpleDateFormat DATE_FORMATTER = new
SimpleDateFormat(DATE_FORMAT);

    private static int NUM_PARAMS = 2;

    private String request;

    private String passengerName;

    private Itinerary itinerary;

    private int clientID;

    private boolean runSuccess;

    /**
     * Create the Reserve command, which creates a reservation for a passenger.
     *
     * @param request the request parameters
     *                for Reserve: id,passenger
     *                where id is the unique ID of the itinerary to reserve,
     *                passenger is the name of the passenger creating the
reservation
     * @param clientID the identification number of the client who made the request
     */
    public Reserve(String request, int clientID) {
        this.request = request;
        this.clientID = clientID;
        this.runSuccess = false;
    }

    /**
```

```
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        runSuccess = false;

        String parseResult = parseRequest();
        if (parseResult != null) {
            return parseResult;
        }

        runSuccess = false;

        ReservationStore reservationStore = ReservationStore.getInstance();
        ReservationStore.Result reserveResult =
reservationStore.reserve(passengerName, itinerary);

        switch (reserveResult) {
            case ALREADY_RESERVED:
                return DUPLICATE;
        }

        runSuccess = true;
        return SUCCESS;
    }

    /**
     * Parse the request string and set the fields with given values.
     *
     * @return null if parse was successful, or an error message to be sent back to
the client
     */
    private String parseRequest() {
        String[] requestArr;
        if (request == null || ((requestArr = request.split(",")).length !=
NUM_PARAMS)) {
            return INVALID_NUMBER_PARAMS;
        }

        int itineraryID;
        try {
            itineraryID = Integer.parseInt(requestArr[0]);
        } catch (NumberFormatException e) {  // NAN
            return INVALID_ID;
        }
        InfoQueryStore infoQueryStore = InfoQueryStore.getInstance();

        if (!infoQueryStore.isValidID(clientID, itineraryID)) {
            return INVALID_ID;
        }

        itinerary = infoQueryStore.getItinerary(clientID, itineraryID);

        passengerName = requestArr[1];

        return null;
    }

    /**
     * Undo this command.
     *
     * @return output to be directed to the client
     */
    @Override
    public String undo() {
        runSuccess = false;

        ReservationStore reservationStore = ReservationStore.getInstance();
        ReservationStore.Result reserveResult = 
reservationStore.delete(passengerName, itinerary);

        switch (reserveResult) {
            case NOT_FOUND:
                return RESERVATION_NOT_FOUND;
        }

        runSuccess = true;

        StringBuilder itineraryString = new StringBuilder();

        List<Flight> flights = itinerary.getFlights();

itineraryString.append(itinerary.getAirfare()).append(",").append(flights.size());

        for (TripComponent flight : flights) {
            itineraryString.append(String.format(FLIGHT_FORMAT, flight.getId(), 
flight.getOrigin().getName(),
                    DATE_FORMATTER.format(flight.getArrival()),
                    flight.getDestination().getName(), 
DATE_FORMATTER.format(flight.getDeparture())));
        }

        return String.format(UNDO_RESP_FORMAT, passengerName,
itineraryString.toString());
    }

    /**
     * Redo this command.
     *
     * @return output to be directed to the client
     */
    @Override
    public String redo() {
        runSuccess = false;

        ReservationStore reservationStore = ReservationStore.getInstance();
        ReservationStore.Result reserveResult = 
reservationStore.reserve(passengerName, itinerary);

        switch (reserveResult) {
            case ALREADY_RESERVED:
                return DUPLICATE;
        }

        runSuccess = true;

        StringBuilder itineraryString = new StringBuilder();

        List<Flight> flights = itinerary.getFlights();

itineraryString.append(itinerary.getAirfare()).append(",").append(flights.size());

        for (TripComponent flight : flights) {
            itineraryString.append(String.format(FLIGHT_FORMAT, flight.getId(), 
flight.getOrigin().getName(),
                    DATE_FORMATTER.format(flight.getArrival()),
                    flight.getDestination().getName(), 
DATE_FORMATTER.format(flight.getDeparture())));
        }

        return String.format(REDO_RESP_FORMAT, passengerName, 
itineraryString.toString());
    }

    /**
     * Did this command run without error on last execution?
     *
     * @return true if the command ran without error, false if any error occurred
     */
    @Override
    public boolean wasSuccessful() {
        return runSuccess;
    }

    /**
     * Get the ID of the client which made this request.
     *
     * @return client ID number
     */
    @Override
    public int getClientID() {
        return clientID;
    }

}
/*
 * File:   Retrieve.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

import model.components.Flight;
import model.components.Itinerary;
import model.components.TripComponent;
import model.data_store.AirportStore;
import model.data_store.ReservationStore;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.List;

public class Retrieve implements Command {

    private static String DATE_FORMAT = "h:mma";

    private static SimpleDateFormat DATE_FORMATTER = new
SimpleDateFormat(DATE_FORMAT);

    private static String FLIGHT_FORMAT = ",%d,%s,%s,%s,%s";

    private static int MIN_PARAMS = 1;

    private static int MAX_PARAMS = 3;

    private static int PASS_IDX = 0;
    private String passenger;
    private static int ORIG_IDX = 1;
    private String origin = "";

    private static int DEST_IDX = 2;
    private String destination = "";

    private String request;

    private int clientID;

    /**
     * Create the Retrieve command, which acquires the reservations for a passenger.
     *
     * @param request the request parameters
     *                for Retrieve: passenger[,origin[,destination]]
     *                where passenger is the name of the passenger to retrieve
reservations for,
     *                origin is the three-letter code of the reservation's origin
airport,
     *                destination is the three-letter code of the reservation's
destination airport
     * @param clientID the identification number of the client who made the request
     */
    public Retrieve(String request, int clientID) {
        this.request = request;
        this.clientID = clientID;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        String parseResult = parseRequest();
        if (parseResult != null) {
            return parseResult;
        }

        ReservationStore reservationStore = ReservationStore.getInstance();
        List<Itinerary> itineraries = reservationStore.retrieve(passenger, origin,
destination);

        StringBuilder output = new StringBuilder("retrieve," + itineraries.size());

        for (Itinerary itinerary : itineraries) {
            output.append("\n");
            List<Flight> flights = itinerary.getFlights();

output.append(itinerary.getAirfare()).append(",").append(flights.size());

            for (TripComponent flight : flights) {
                output.append(String.format(FLIGHT_FORMAT, flight.getId(),
flight.getOrigin().getName(),
                        DATE_FORMATTER.format(flight.getArrival()),
                        flight.getDestination().getName(),
DATE_FORMATTER.format(flight.getDeparture())));
            }

        }

        return output.toString();
    }

    /**
     * Parse the request string and set the fields with given values.
     *
     * @return null if parse was successful, or an error message to be sent back to
the client
     */
    private String parseRequest() {
        // check number parameters
        String[] requestArr;
        if (request == null ||
                ((requestArr = request.split(",")).length > MAX_PARAMS) ||
requestArr.length < MIN_PARAMS) {
            return INVALID_NUMBER_PARAMS;
        }

        passenger = requestArr[PASS_IDX];

        // check valid origin and destination
        AirportStore airportStore = AirportStore.getInstance();

        if (requestArr.length > MIN_PARAMS) {
            origin = requestArr[ORIG_IDX];

            // origin may be blank when destination is given
            if (!origin.equals("")) {
                if (!airportStore.isAirport(origin)) {
                    return UNKNOWN_ORIG;
                }
            }
        }
        if (requestArr.length > MIN_PARAMS + 1) {
            destination = requestArr[DEST_IDX];
            if (!airportStore.isAirport(destination)) {
                return UNKNOWN_DEST;
            }
        }

        return null;
    }
}

package control.request_response.commands;

import model.data_store.ServerStore;

public class Server implements Command {
    private int cid;
    private String request;
    private String server;
    private String local = "local";
    private String faa = "faa";
    private String UNKNOWN_SERVER = "error,unknown request";
    public Server(String request, int clientID) {
        this.request = request;
        this.cid = clientID;
    }
    @Override
    public String execute() {
        String parseResult = parseRequest();
        ServerStore instance = ServerStore.getInstance();
        if(parseResult != null) {
            return parseResult;
        }
        if(server.toLowerCase().equals(local)) {

            instance.changeServer(cid, ServerStore.Server.local);

//control.request_response.commands.Airport.setServer(control.request_response.comma
nds.Airport.Server.LOCAL);
            return String.format("changed server to " + local);

        }
        if(server.toLowerCase().equals(faa)) {
            instance.changeServer(cid, ServerStore.Server.faa);

//control.request_response.commands.Airport.setServer(control.request_response.comma
nds.Airport.Server.FAA);
            return String.format("changed server to " + faa);
        }

        return null;
    }
    private String parseRequest() {
        if(request == null || request.split(",").length > 1) {
            return INVALID_NUMBER_PARAMS;
        }

        server = request;

        if(!((server.toLowerCase().equals(local)) ||
((server.toLowerCase().equals(faa))))) {
            return UNKNOWN_SERVER;
        }

        return null;


    }

}
package control.request_response.commands;

/**
 * The undo command allows UndoManager to know to undo a command.
 */
public class Undo implements Command {

    private boolean validUndo;

    private int clientID;

    /**
     * Create the undo command.
     *
     * @param request the request parameters, null if no params
     * @param clientID ID number of client which made request
     */
    public Undo(String request, int clientID) {
        this.clientID = clientID;
        this.validUndo = request == null;  // undo command should have no request
parameters
    }

    /**
     * Was this undo request valid?
     *
     * @return true if the undo request was valid, false if not
     */
    public boolean isValid() {
        return validUndo;
    }

    /**
     * Get the ID of the client which made the undo request.
     *
     * @return client ID number
     */
    public int getClientID() {
        return clientID;
    }

    /**
     * Complete this command's operation.
     *
     * @return the response to be given back to the client
     */
    @Override
    public String execute() {
        if (!validUndo) {
            return INVALID_NUMBER_PARAMS;
        }
        return null;  // result of un-done command will be response, so this return 
is ignored
    }
}
/*
 * File:   UndoableCommand.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response.commands;

/**
 * Interface defining functionality for a command which is undoable and redoable.
 * Commands which make a state change to the system are good candidates to be 
undoable and redoable.
 */
public interface UndoableCommand extends Command {

    /**
     * Did this command run without error on last execution.
     *
     * @return true if the command ran without error, false if any error occurred
     */
    boolean wasSuccessful();

    /**
     * Get the ID of the client which made this request.
     *
     * @return client ID number
     */
    int getClientID();

    /**
     * Undo this command.
     *
     * @return output to be directed to the client
     */
    String undo();

    /**
     * Redo this command.
     *
     * @return output to be directed to the client
     */
    String redo();

}
/*
 * File:   ClientIdentifierManager.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response;

import java.util.ArrayList;
import java.util.List;
import java.util.TreeSet;

/**
 * Provides and keeps track of client identification numbers.
 */
public class ClientIdentifierManager implements ClientIdentifierSubject {

    private List<ClientIdentifierObserver> observers = new ArrayList<>();

    private TreeSet<Integer> activeIDs = new TreeSet<>();

    private TreeSet<Integer> availableIDs = new TreeSet<>();
    private static int MAX_EXTRA_IDS = 10;

    /**
     * Register a new client. A unique client identification number will be provided
for use until the client
     * disconnects.
     *
     * @return unique client ID to represent the new client
     */
    public int registerNewClient() {
        int id;

        if (availableIDs.isEmpty()) {
            if (!activeIDs.isEmpty()) {
                id = activeIDs.last() + 1;
            } else {
                id = 1;  // => lowest and first ID is 1
            }
        } else {
            id = availableIDs.pollFirst();  // get the lowest available ID
        }

        activeIDs.add(id);

        notifyObserversNewID(id);

        return id;
    }

    /**
     * Release the identification number associated with a client. This should be
done when a client disconnects from the
     * system.
     *
     * @param clientID the client ID number to be released
     */
    public void releaseClient(int clientID) {
        if (activeIDs.contains(clientID)) {
            activeIDs.remove(clientID);
            availableIDs.add(clientID);
            notifyObserversReleasedID(clientID);
        }

        while (availableIDs.size() > MAX_EXTRA_IDS) {  // control growth of IDs
            availableIDs.pollLast();
        }
    }
    /**
     * Check if a given client ID is in use by a client.
     *
     * @param clientID client identification number to check
     * @return true if the client ID is in use, false if not
     */
    public boolean isActiveClient(int clientID) {
        return activeIDs.contains(clientID);
    }

    /**
     * Attach a new observer to this subject.
     *
     * @param observer the observer to attach
     */
    @Override
    public void attachObserver(ClientIdentifierObserver observer) {
        observers.add(observer);
    }

    /**
     * Notify all observers of a new client ID being registered.
     *
     * @param newID the new client ID number
     */
    private void notifyObserversNewID(int newID) {
        for (ClientIdentifierObserver observer: observers) {
            observer.notifyNewClient(newID);
        }
    }

    /**
     * Notify all observers that a client ID was released.
     *
     * @param releasedID the released ID number
     */
    private void notifyObserversReleasedID(int releasedID) {
        for (ClientIdentifierObserver observer: observers) {
            observer.notifyClientDisconnect(releasedID);
        }
    }

}
/*
 * File:   ClientIdentifierObserver.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response;

/**
 * Interface defining methods required by observers of the ClientIDAssigner.
 */
public interface ClientIdentifierObserver {

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    void notifyNewClient(int clientID);

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    void notifyClientDisconnect(int clientID);

}
/*
 * File:   ClientIdentifierSubject.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response;

/**
 * Interface defining the methods to attach a ClientIdentifierObserver to the
ClientIdentifierManager.
 */
public interface ClientIdentifierSubject {

    /**
     * Attach a new observer to this subject.
     *
     * @param observer the observer to attach
     */
    void attachObserver(ClientIdentifierObserver observer);

}
/*
 * File:   CommandFactory.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response;

import control.request_response.commands.*;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Map;

/**
 * Class responsible for the creation of Commands. The type of Command created is
based on a keyword contained in
 * a client request.
 */
public class CommandFactory {

    /**
     * Mapping of request keywords to their respective Command class.
     */
    static final Map<String, Class> commandMap = new HashMap<String, Class>() {
        {
            put("info", Info.class);
            put("reserve", Reserve.class);
            put("retrieve", Retrieve.class);
            put("delete", Delete.class);
            put("airport", Airport.class);
            put("server", Server.class);
            put("undo", Undo.class);
            put("redo", Redo.class);
        }
    };

    /**
     * Create a command object based on a request's keyword.
     *
     * @param keyword the request keyword, which should exist in the command map.
Case does not matter.
     * @param request the full request (not including keyword)
     * @param clientID identification number of the client who sent this request
     * @return the new Command object, null if bad keyword
     */
    public Command createCommand(String keyword, String request, int clientID)
            throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException, InstantiationException {
        keyword = keyword.toLowerCase();  // keywords are in map lower case
        if (!commandMap.containsKey(keyword)) {
            return null;
        }

        // use reflection to get the constructor for the command
        Class<?> commandType = commandMap.get(keyword);
        Class<?>[] paramTypes = new Class[] {String.class, int.class};
        Constructor<?> ctor;
        try {
            ctor = commandType.getConstructor(paramTypes);
        } catch (NoSuchMethodException e) {
            System.err.println("Command class constructor not found for keyword: " + 
keyword);
            throw e;
        }

        // create the new command from the constructor
        Command command;
        try {
             command = (Command)ctor.newInstance(request, clientID);
        } catch (InstantiationException | IllegalAccessException | 
InvocationTargetException e) {
            e.printStackTrace();
            System.err.println("Could not create instance of command for keyword: " 
+ keyword);
            throw e;
        }

        return command;
    }

}
/*
 * File:   RequestHandler.java
 * Author: Adam Del Rosso
 * Email:  avd5772@rit.edu
 * GitHub: AdamVD
 */
package control.request_response;

import control.request_response.commands.Command;

import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Map;

/**
```

```
 * The RequestHandler is the entry point for the AFRS system, receiving and
responding to client's text-based API calls.
 */
public class RequestHandler implements ClientIdentifierObserver {

    private static String RESPONSE_FORMAT = "%d,%s";  // CID,response text

    static String PARTIAL_REQUEST_MSG = "partial-request";

    static String INTERNAL_ERROR = "error,internal error occurred";

    static String UNKNOWN_KEYWORD = "error,unknown request";

    private static String BAD_CID = "error,invalid connection";

    private static String CONNECT_RESPONSE = "connect,%d";

    private static String CONNECT_RQST_KW = "connect";

    private static String DISCONNECT_RESPONSE = "%d,disconnect";

    private static String DISCONNECT_KW = "disconnect";

    private CommandFactory commandFactory;

    private Map<Integer, String> incompleteRequestMap;

    private ClientIdentifierManager clientIdentifierManager;

    private UndoManager undoManager;

    /**
     * Create a request handler.
     */
    public RequestHandler(CommandFactory commandFactory, ClientIdentifierManager
clientIdentifierManager,
                          UndoManager undoManager) {
        this.commandFactory = commandFactory;
        this.incompleteRequestMap = new HashMap<>();
        this.clientIdentifierManager = clientIdentifierManager;
        this.undoManager = undoManager;
    }

    /**
     * Make a request to the AFRS system.
     *
     * @param request the request, which will only be fully processed once closed
with a semi-colin
```

```
      * @return response to the request
      */
     public String makeRequest(String request) {
         if (!request.contains(";")) {  // complete requests end in a semi-colin
             int cid;
             try {
                 String[] cidSplit = request.split(",",2);  // breaks into [CID,
partial-request]
                 cid = Integer.parseInt(cidSplit[0]);  // CID should be first item in
CSV

                 if (clientIdentifierManager.isActiveClient(cid)) {
                     incompleteRequestMap.put(cid, incompleteRequestMap.get(cid) +
cidSplit[1]);
                     return PARTIAL_REQUEST_MSG;
                 }
             } catch (NumberFormatException e) {  // NAN
                 return BAD_CID;
             }

             return BAD_CID;
         }

         request = request.replace(";", "");  // we don't care about the ending
semi-colin

         if (request.trim().equalsIgnoreCase(CONNECT_RQST_KW)) {  // new client
             int id = clientIdentifierManager.registerNewClient();
             incompleteRequestMap.put(id, "");
             return String.format(CONNECT_RESPONSE, id);
         }

         int cid;
         String[] cidSplit;
         try {
             cidSplit = request.split(",",2);  // breaks into [CID, partial-request]
             cid = Integer.parseInt(cidSplit[0]);  // CID should be first item in CSV

             if (!clientIdentifierManager.isActiveClient(cid)) {
                 return BAD_CID;
             }
         } catch (NumberFormatException e) {  // NAN
             return BAD_CID;
         }

         request = incompleteRequestMap.get(cid) + cidSplit[1];

         // reset incomplete for the client
```

```
        incompleteRequestMap.remove(cid);
        incompleteRequestMap.put(cid, "");

        return processCompleteRequest(cid, request);
    }

    /**
     * Process a complete request.
     *
     * @param CID the client identification number of the client which sent the
request
     * @param request a full request (not necessarily valid)
     * @return the response for the client
     */
    private String processCompleteRequest(int CID, String request) {
        String[] kwParamsSplit = request.split(",", 2);  // splits into [kw, params]
        String keyword = kwParamsSplit[0];

        if (kwParamsSplit.length > 1) {
            request = kwParamsSplit[1];  // commands want only the request
parameters
        } else {
            request = null;  // null => no parameters
        }

        if (keyword.equalsIgnoreCase(DISCONNECT_KW) && request == null) {
            return handleDisconnectRequest(CID);
        }

        Command command;
        try {
            command = commandFactory.createCommand(keyword, request, CID);
        } catch (NoSuchMethodException | IllegalAccessException |
InvocationTargetException | InstantiationException e) {
            // these errors come from the reflection code within command factory
            e.printStackTrace();
            System.err.println("If a new command was added, check that it follows
the same constructor constraints as" +
                    "other command classes. Failed with keyword: " + keyword);
            return INTERNAL_ERROR;
        }

        if (command == null) {  // command == null => keyword is unknown by the
command factory
            return UNKNOWN_KEYWORD;
        }

        try {
            String response = undoManager.executeCommand(command);
            return String.format(RESPONSE_FORMAT, CID, response);
        } catch (Exception e) {  // catch any error that made it all the way out so
that the system does not stop
            e.printStackTrace(System.err);
            return INTERNAL_ERROR;
        }
    }

    /**
     * Handle a request to disconnect.
     *
     * @param CID client identification number of the client wishing to disconnect
     */
    private String handleDisconnectRequest(int CID) {
        clientIdentifierManager.releaseClient(CID);
        return String.format(DISCONNECT_RESPONSE, CID);
    }

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    @Override
    public void notifyNewClient(int clientID) {
        incompleteRequestMap.put(clientID, "");
    }

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    @Override
    public void notifyClientDisconnect(int clientID) {
        incompleteRequestMap.remove(clientID);
    }
}
package control.request_response;

import control.request_response.commands.Command;
import control.request_response.commands.Redo;
import control.request_response.commands.Undo;
import control.request_response.commands.UndoableCommand;

import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

/**
 * UndoManager is responsible for storing all UndoableCommands which were 
successfully executed.
 * All commands come through the UndoManager, which executes them and returns the 
response to the RequestHandler for
 * client output.
 */
public class UndoManager implements ClientIdentifierObserver {

    private static String NONE_TO_UNDO = "error,no request available";

    private Map<Integer, Stack<UndoableCommand>> undoMap;

    private Map<Integer, Stack<UndoableCommand>> redoMap;

    /**
     * Create an UndoManager. There should only be one in the system.
     *
     * @param IDmanager the client ID manager
     */
    public UndoManager(ClientIdentifierSubject IDmanager) {
        IDmanager.attachObserver(this);
        this.undoMap = new HashMap<>();
        this.redoMap = new HashMap<>();
    }

    /**
     * Execute a command and store it for undo/redo is a working UndoableCommand.
     *
     * @param command the command to execute
     * @return output of the command to be returned to the client
     */
    public String executeCommand(Command command) {
        String response = command.execute();

        if (command instanceof Undo && ((Undo) command).isValid()) {  // valid undo 
given, undo last for client
            int clientID = ((Undo) command).getClientID();

            if (undoMap.get(clientID).isEmpty()) {
                response = NONE_TO_UNDO;
            } else {
                response = undoLast(clientID);
            }
        } else if (command instanceof Redo && ((Redo) command).isValid()) {  // 
valid redo given, redo last
```

```
            int clientID = ((Redo) command).getClientID();

            if (redoMap.get(clientID).isEmpty()) {
                response = NONE_TO_UNDO;
            } else {
                response = redoLast(clientID);
            }
        }

        // this command should be stored if it is an UndoableCommand and executed
successfully
        if (command instanceof UndoableCommand && ((UndoableCommand)
command).wasSuccessful()) {
            UndoableCommand undoableCommand = (UndoableCommand) command;
            int clientID = undoableCommand.getClientID();
            undoMap.get(clientID).push(undoableCommand);
        }

        return response;
    }

    private String undoLast(int clientID) {
        UndoableCommand toUndo = undoMap.get(clientID).pop();
        String response = toUndo.undo();
        if (toUndo.wasSuccessful()) {  // only keep if successful
            redoMap.get(clientID).push(toUndo);
        }
        return response;
    }

    private String redoLast(int clientID) {
        UndoableCommand toRedo = redoMap.get(clientID).pop();
        String response = toRedo.redo();
        if (toRedo.wasSuccessful()) {  // only keep if successful
            undoMap.get(clientID).push(toRedo);
        }
        return response;
    }

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    @Override
    public void notifyNewClient(int clientID) {
        undoMap.put(clientID, new Stack<>());
        redoMap.put(clientID, new Stack<>());
    }

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    @Override
    public void notifyClientDisconnect(int clientID) {
        undoMap.remove(clientID);
        undoMap.remove(clientID);
    }
}
package model.components;

import java.util.Comparator;

public class AirfareComparator implements Comparator<TripComponent> {
    /**
     * Compares its two arguments for order.  Returns a negative integer,
     * zero, or a positive integer as the first argument is less than, equal
     * to, or greater than the second.<p>
     * <p>
     * In the foregoing description, the notation
     * <tt>sgn(</tt><i>expression</i><tt>)</tt> designates the mathematical
     * <i>signum</i> function, which is defined to return one of <tt>-1</tt>,
     * <tt>0</tt>, or <tt>1</tt> according to whether the value of
     * <i>expression</i> is negative, zero or positive.<p>
     * <p>
     * The implementor must ensure that <tt>sgn(compare(x, y)) ==
     * -sgn(compare(y, x))</tt> for all <tt>x</tt> and <tt>y</tt>.  (This
     * implies that <tt>compare(x, y)</tt> must throw an exception if and only
     * if <tt>compare(y, x)</tt> throws an exception.)<p>
     * <p>
     * The implementor must also ensure that the relation is transitive:
     * <tt>((compare(x, y)&gt;0) &amp;&amp; (compare(y, z)&gt;0))</tt> implies
     * <tt>compare(x, z)&gt;0</tt>.<p>
     * <p>
     * Finally, the implementor must ensure that <tt>compare(x, y)==0</tt>
     * implies that <tt>sgn(compare(x, z))==sgn(compare(y, z))</tt> for all
     * <tt>z</tt>.<p>
     * <p>
     * It is generally the case, but <i>not</i> strictly required that
     * <tt>(compare(x, y)==0) == (x.equals(y))</tt>.  Generally speaking,
     * any comparator that violates this condition should clearly indicate
     * this fact.  The recommended language is "Note: this comparator
     * imposes orderings that are inconsistent with equals."
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as the
     * first argument is less than, equal to, or greater than the
     * second.
     * @throws NullPointerException if an argument is null and this
     *                             comparator does not permit null arguments
     * @throws ClassCastException   if the arguments' types prevent them from
     *                             being compared by this comparator.
     */
    @Override
    public int compare(TripComponent o1, TripComponent o2) {
        if (o1.getAirfare() < o2.getAirfare()) {
            return -1;
        } else if (o1.getAirfare() > o2.getAirfare()) {
            return 1;
        } else {
            return 0;
        }
    }
}
/**
 * Airport class
 * @author Meet Patel
 */
package model.components;

import control.request_response.ClientIdentifierObserver;
import control.request_response.ClientIdentifierSubject;

import java.util.*;

public class Airport implements ClientIdentifierObserver {


    String name;
    String code;
    List<String> weather;
    int delayTime;
    int connectionTime;

    Map<Integer, Iterator<String>> clientWeather;

    public Airport (ClientIdentifierSubject clientIDManager) {
        this.clientWeather = new HashMap<>();
        clientIDManager.attachObserver(this);
    }
```

```
public String getName() {
    return name;
}

public String getCode() {
    return code;
}

/**
 * Get the current weather at this airport.
 *
 * @return weather in the format "condition,temperature"
 */
public String getWeather(int CID) {
    if (!(clientWeather.containsKey(CID) && clientWeather.get(CID).hasNext())) {
        clientWeather.put(CID, weather.iterator());
    }
    return clientWeather.get(CID).next();
}

public int getDelayTime() {
    return delayTime;
}

public int getConnectionTime() {
    return connectionTime;
}
public void setName(String name) {
    this.name = name;
}
public void setCode(String code) {
    this.code = code;
}
public void setWeather(List<String> weather) {
    this.weather = weather;
}
public void setDelayTime(int delayTime) {
    this.delayTime = delayTime;
}
public void setConnectionTime(int connectionTime) {
    this.connectionTime = connectionTime;
}



@Override
public String toString() {
    return super.toString();
    }

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    @Override
    public void notifyNewClient(int clientID) {
        // do nothing... putting every client and new weather iterator into the map 
at each airport is a waste of memory
        // in this case memory cost > time savings
    }

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    @Override
    public void notifyClientDisconnect(int clientID) {
        clientWeather.remove(clientID);
    }
}
package model.components;

import java.util.Comparator;

public class ArrivalComparator implements Comparator<TripComponent> {
    /**
     * Compares its two arguments for order.  Returns a negative integer,
     * zero, or a positive integer as the first argument is less than, equal
     * to, or greater than the second.<p>
     * <p>
     * In the foregoing description, the notation
     * <tt>sgn(</tt><i>expression</i><tt>)</tt> designates the mathematical
     * <i>signum</i> function, which is defined to return one of <tt>-1</tt>,
     * <tt>0</tt>, or <tt>1</tt> according to whether the value of
     * <i>expression</i> is negative, zero or positive.<p>
     * <p>
     * The implementor must ensure that <tt>sgn(compare(x, y)) ==
     * -sgn(compare(y, x))</tt> for all <tt>x</tt> and <tt>y</tt>.  (This
     * implies that <tt>compare(x, y)</tt> must throw an exception if and only
     * if <tt>compare(y, x)</tt> throws an exception.)<p>
     * <p>
     * The implementor must also ensure that the relation is transitive:
     * <tt>((compare(x, y)&gt;0) &amp;&amp; (compare(y, z)&gt;0))</tt> implies
     * <tt>compare(x, z)&gt;0</tt>.<p>
     * <p>
     * Finally, the implementor must ensure that <tt>compare(x, y)==0</tt>
     * implies that <tt>sgn(compare(x, z))==sgn(compare(y, z))</tt> for all
     * <tt>z</tt>.<p>
     * <p>
     * It is generally the case, but <i>not</i> strictly required that
     * <tt>(compare(x, y)==0) == (x.equals(y))</tt>.  Generally speaking,
     * any comparator that violates this condition should clearly indicate
     * this fact.  The recommended language is "Note: this comparator
     * imposes orderings that are inconsistent with equals."
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as the
     * first argument is less than, equal to, or greater than the
     * second.
     * @throws NullPointerException if an argument is null and this
     *                              comparator does not permit null arguments
     * @throws ClassCastException   if the arguments' types prevent them from
     *                              being compared by this comparator.
     */
    @Override
    public int compare(TripComponent o1, TripComponent o2) {
        return o1.getArrival().compareTo(o2.getArrival());
    }
}
package model.components;

import java.util.Comparator;

public class DepartureComparator implements Comparator<TripComponent> {
    /**
     * Compares its two arguments for order.  Returns a negative integer,
     * zero, or a positive integer as the first argument is less than, equal
     * to, or greater than the second.<p>
     * <p>
     * In the foregoing description, the notation
     * <tt>sgn(</tt><i>expression</i><tt>)</tt> designates the mathematical
     * <i>signum</i> function, which is defined to return one of <tt>-1</tt>,
     * <tt>0</tt>, or <tt>1</tt> according to whether the value of
     * <i>expression</i> is negative, zero or positive.<p>
     * <p>
     * The implementor must ensure that <tt>sgn(compare(x, y)) ==
     * -sgn(compare(y, x))</tt> for all <tt>x</tt> and <tt>y</tt>.  (This
     * implies that <tt>compare(x, y)</tt> must throw an exception if and only
     * if <tt>compare(y, x)</tt> throws an exception.)<p>
     * <p>
     * The implementor must also ensure that the relation is transitive:
```

```
     * <tt>((compare(x, y)&gt;0) &amp;&amp; (compare(y, z)&gt;0))</tt> implies
     * <tt>compare(x, z)&gt;0</tt>.<p>
     * <p>
     * Finally, the implementor must ensure that <tt>compare(x, y)==0</tt>
     * implies that <tt>sgn(compare(x, z))==sgn(compare(y, z))</tt> for all
     * <tt>z</tt>.<p>
     * <p>
     * It is generally the case, but <i>not</i> strictly required that
     * <tt>(compare(x, y)==0) == (x.equals(y))</tt>.  Generally speaking,
     * any comparator that violates this condition should clearly indicate
     * this fact.  The recommended language is "Note: this comparator
     * imposes orderings that are inconsistent with equals."
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as the
     * first argument is less than, equal to, or greater than the
     * second.
     * @throws NullPointerException if an argument is null and this
     *                              comparator does not permit null arguments
     * @throws ClassCastException   if the arguments' types prevent them from
     *                              being compared by this comparator.
     */
    @Override
    public int compare(TripComponent o1, TripComponent o2) {
        return o1.getDeparture().compareTo(o2.getDeparture());
    }
}
/**
 * Flight Class
 * @author Meet Patel
 */

package model.components;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.TimeUnit;

public class Flight implements TripComponent {
    private Airport origin;
    private Airport destination;
    private Date arrival;
    private Date departure;
    private int airfare;
    private int flightNum;
    /**
     * @param origin: Airport flight starts at
     * @param destination airport flight lands at
     * @param arrivalStr time the flight starts as, turned into a date object by
this constructor
     * @param departureStr time the flight ends as, turned into a date object by
this constructor
     * @param airfare how much the flight costs
     * @param flightNum flights unique id
     */
    public Flight(Airport origin, Airport destination, String arrivalStr, String
departureStr, int flightNum, int airfare) throws ParseException{

        this.origin = origin;
        this.destination = destination;
        this.airfare = airfare;
        this.flightNum = flightNum;
        arrivalStr += "m";
        departureStr += "m";
        DateFormat form = new SimpleDateFormat("hh:mma");
        arrival = form.parse(arrivalStr);
        departure = form.parse(departureStr);
    }

    public Airport getOrigin() {
        return origin;
    }

    public Airport getDestination() {
        return destination;
    }

    @Override
    public int getId() {
        return flightNum;
    }

    @Override
    public Date getArrival() {
        return arrival;
    }

    @Override
    public Date getDeparture(){
        return departure;
    }
    public int getAirfare() {
        return airfare;
    }

    public int getFlightNum() {
        return flightNum;
    }


    /**
     * Shows the differance between the arrival time and the departure time
     * @return The time between arrival and departure
     */
    public int getTime() throws ParseException {
        long diffInMillies = arrival.getTime() - departure.getTime();
        int diffInMinutes = (int)TimeUnit.MINUTES.convert(diffInMillies, 
TimeUnit.MILLISECONDS);
        return diffInMinutes;
    }


}
/**
 * Itinerary Class
 * @author Meet Patel
 */
package model.components;

import java.text.ParseException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Itinerary implements TripComponent {

    List<Flight> flights;
    int id;
    int connections;

    public Itinerary(List<Flight> flights) {
        this.flights = flights;
        connections = flights.size();
    }

    /**
     * Calls getTime on all of the child classes of the Itinerary
     * @return sum of all of the child class' time.
     * @throws ParseException
     */
    @Override
    public int getTime() throws ParseException {
        int res = 0;
        for (TripComponent f : flights) {
            res += f.getTime();
        }
        return res;
    }

    /**
     * Calls getAirfare on all of the child classes of the Itinerary
     * @return sum of all of the child class' airfare.
     * @throws ParseException
     */
    @Override
    public int getAirfare() {
        int res = 0;
        for (TripComponent f : flights) {
            res += f.getAirfare();
        }
        return res;
    }

    /**
     * Remove a flight from an itinerary
     * @throws Exception
     */
    public void remove(TripComponent f) throws Exception {
        if (flights.contains(f)) {
            flights.remove(f);
        } else {
            throw new Exception("This flight is not in the itinerary");
        }
    }

    public List<Flight> getFlights() {
        return flights;
    }

    public int getId() {
        return id;
    }

    @Override
    public Airport getOrigin() {
        return flights.get(0).getOrigin();
    }
    @Override
    public Airport getDestination() {
        return flights.get(flights.size() - 1).getDestination();
    }

    @Override
    public Date getArrival() {
        return flights.get(flights.size()-1).getArrival();
    }

    @Override
    public Date getDeparture() {
        return flights.get(0).getDeparture();
    }
    public List<Integer> getFlightIntegers() {
        List<Integer> flightNums = new ArrayList<Integer>();
        for(int i = 0; i < flights.size(); i++) {
            flightNums.add(flights.get(i).getId());
        }
        return flightNums;
    }

}
/**
 * TripCompenent interface
 * @author Meet Patel
 */

package model.components;

import java.text.ParseException;
import java.util.Date;

public interface TripComponent {

    int getTime() throws ParseException;
    int getAirfare();
    Airport getOrigin();
    Airport getDestination();
    Date getArrival();
    Date getDeparture();
    int getId();
}
package model.data_store;
```

```
public interface AirportData {
    String getAirportName(String code);
    String getAirportWeather(int clientID, String code);
    String getAirportDelay(String code);
}
/**
 * @author: Shawn Struble
 */
package model.data_store;

import control.request_response.ClientIdentifierManager;
import control.request_response.ClientIdentifierSubject;
import model.components.Airport;

import java.io.*;
import java.util.*;

public class AirportStore implements AirportData {

    private ClientIdentifierSubject clientIDManager;

    private static AirportStore instance = null;
    private List<Airport> airports = new ArrayList<Airport>();
    private static String airportsPath;
    private static String airportWeatherPath;
    private static String airportConnectionsPath;
    private static String airportDelayPath;
    public AirportStore(String airportsPath,String airportWeatherPath, String 
airportConnectionsPath,
                        String airportDelay, ClientIdentifierSubject 
clientIDManager) {
        this.airportsPath = airportsPath;
        this.airportWeatherPath = airportWeatherPath;
        this.airportConnectionsPath = airportConnectionsPath;
        this.airportDelayPath = airportDelay;
        this.clientIDManager = clientIDManager;

        readAirports();
        readAirPortWeather();
        readAirportDelay();
        readAirPortConnections();
    }
    /**
     *
     * @return the singleton instance of AirportStore
     * throws an error if instance was not created with file paths
     */
    public static AirportStore getInstance() {
        if (instance == null) {
            throw new RuntimeException("AirportStore not properly instantiated!");
        }
        return instance;
    }

    /**
     * creates instance with all file paths
     * @param airportsPath
     * @param airportWeatherPath
     * @param airportConnectionsPath
     * @param airportDelay
     * @return the singleton instance of a class
     */
    public static AirportStore getInstance(String airportsPath, String 
airportWeatherPath, String airportConnectionsPath,
                                           String airportDelay, 
ClientIdentifierManager clientIDManager) {
        if (instance == null) {
            instance = new 
AirportStore(airportsPath,airportWeatherPath,airportConnectionsPath,airportDelay, 
clientIDManager);
        }
        return instance;

    }
    /**
     * Populates the airports list
     * TODO: Add check to make sure same airport doesn't get added twice?
     */
    private void readAirports() {
        String line = "";
        try (BufferedReader br = new BufferedReader(new FileReader(airportsPath))){
            while((line = br.readLine()) != null) {
                Airport airport = new Airport(clientIDManager);
                String[] airportInfo = line.split(",");
                airport.setCode(airportInfo[0]); // Airport Code Ex. "ATL"
                airport.setName(airportInfo[1]); // Name Ex. "Atlanta"
                airports.add(airport);

            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();

        }

    }
    /**
     * reads in airport delays from file and adds them to their respective objects
     */
    private void readAirportDelay() {
        String line = "";
        try(BufferedReader br = new BufferedReader(new 
FileReader(airportDelayPath))) {
            int i = 0;
            while((line = br.readLine()) != null) {
                String[] airportDelayInfo = line.split(",");
                Airport airport = airports.get(i);
                airport.setDelayTime(Integer.parseInt(airportDelayInfo[1]));
                i++;
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();

        }

    }
    /**
     * reads in airport connection time from file and adds them to their respective 
objects
     */
    private void readAirPortConnections() {
        String line = "";
        try(BufferedReader br = new BufferedReader(new 
FileReader(airportConnectionsPath))) {
            int i = 0;
            while((line = br.readLine()) != null) {
                String[] connectionInfo = line.split(",");
                Airport airport = airports.get(i);
                airport.setConnectionTime(Integer.parseInt(connectionInfo[1]));
                i++;
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Adds weather info to airport objects
     * TODO: finish this method
     */
    private void readAirPortWeather() {
        String line = "";
        try (BufferedReader br = new BufferedReader(new 
FileReader(airportWeatherPath))) {
            int count = 0;
            while ((line = br.readLine()) != null) {
                ArrayList<String> weatherConditions= new ArrayList<String>();
                String[] weatherInfo = line.split(",");
                for (int i = 1; i < weatherInfo.length-1; i += 2) {
                    weatherConditions.add(weatherInfo[i] + "," + weatherInfo[i+1]);
                }
                airports.get(count).setWeather(weatherConditions);
                count++;
            }
            count++;
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    /**
     * finds airport object in airports arraylist based on 3 digit airport code
     * @param airportCode - used to find airport object
     * @return - airport object
     * returns null on failure
     */
     Airport getAirportObject(String airportCode) {
        airportCode = airportCode.toUpperCase();
        Airport returnPort;
        for(int i = 0; i < airports.size(); i++) {
            if(airportCode.equals(airports.get(i).getCode())) {
                returnPort = airports.get(i);
                return returnPort;
            }
        }
        return null;

    }

    /**
     * Checks if the provided code is a three letter code for an airport in the
system.
     *
     * @param code potential airport code to check
     * @return true if the code is valid for an airport, false if not a valid code
or no airport found
     */
    public boolean isAirport(String code) {
        Airport airport = getAirportObject(code);
        return airport != null;
    }

    /**
     * Get the full name of an airport.
     *
     * @param code the three letter code of an airport
     *      Note: this code is assumed valid, check with isAirport before calling
     * @return full name of the airport
     */
    public String getAirportName(String code) {
        Airport airport = getAirportObject(code);
        return airport.getName();
    }

    /**
     * Get the current weather at an airport.
     *
     * @param code the three letter code of an airport
     *      Note: this code is assumed valid, check with isAirport before calling
     * @return current weather at airport
     */
    public String getAirportWeather(int CID, String code) {
        Airport airport = getAirportObject(code);
        return airport.getWeather(CID);
    }

    /**
     * Get the delay at an airport.
     *
     * @param code the three letter code of an airport
     *      Note: this code is assumed valid, check with isAirport before calling
     * @return delay at airport
     */
    public String getAirportDelay(String code) {
        Airport airport = getAirportObject(code);
        return "" + airport.getDelayTime();
    }
}
package model.data_store;

import com.google.gson.JsonArray;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

import javax.xml.ws.ProtocolException;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class FAAAirport implements AirportData {
    private static FAAAirport instance = null;
    private static AirportStore localInstance = null;
    private JsonParser parser;
    private String urlPreface =
"https://soa.smext.faa.gov/asws/api/airport/status/";
    public static FAAAirport getInstance() {
        if(instance == null) {
            instance = new FAAAirport();
        }
        return instance;
    }
    private FAAAirport() {
        localInstance = AirportStore.getInstance();
       parser = new JsonParser();
    }
    @Override
    public String getAirportName(String code) {

        return localInstance.getAirportName(code);
     }

    @Override
    public String getAirportWeather(int ClientID, String code) {
        try {
            String response = responseGen(code);

            JsonObject rootObj = parser.parse(response).getAsJsonObject();

            JsonObject weather1 = rootObj.getAsJsonObject("Weather");
            JsonArray weather2 = weather1.getAsJsonArray("Weather");

            JsonObject temperatureObj = weather2.get(0).getAsJsonObject();



            String actualTemp = weather1.get("Temp").getAsString();


            String tempDesc = temperatureObj.get("Temp").getAsString();

            String returnString = tempDesc + "," + actualTemp;
            return returnString;
        } catch (Exception e) {
            e.printStackTrace();
        }

        return null;
    }

    @Override
    public String getAirportDelay(String code) {
        try {
            String response = responseGen(code);
            JsonObject rootObj = parser.parse(response).getAsJsonObject();

            JsonArray status = rootObj.get("Status").getAsJsonArray();

            String avgDelay = "0";
            JsonObject delay = status.get(0).getAsJsonObject();

            if(delay.has("Reason") == false) {
                if(delay.has("AvgDelay" )) {
                    avgDelay = delay.get("AvgDelay").getAsString();
                }
                //Just In Case
                else if (delay.has("avgDelay" )) {
                    avgDelay = delay.get("avgDelay").getAsString();
                }
                else if(delay.has("MaxDelay")){
                    avgDelay = delay.get("MaxDelay").getAsString();
                }
                else if(delay.has("MinDelay")){
                    avgDelay = delay.get("MinDelay").getAsString();
                }
                else {
                    avgDelay = "Error,FAA found no delay time";
                }
            }
            return avgDelay;


        } catch(Exception e){
            e.printStackTrace();

        }
        return "0";
    }
    private String responseGen(String airport) throws Exception{
        String url;

        StringBuilder response = new StringBuilder();


        try {        // Create a URL and open a connection
            url = urlPreface + airport;
            URL FAAURL = new URL(url);
            HttpURLConnection urlConnection =
                    (HttpURLConnection) FAAURL.openConnection();

            // Set the paramters for the HTTP Request
            urlConnection.setRequestMethod("GET");
            urlConnection.setConnectTimeout(10000);
            urlConnection.setReadTimeout(10000);
            urlConnection.setRequestProperty("Accept", "application/json");



            // Create an input stream and wrap in a BufferedReader to read the
            // response.
            BufferedReader in =
                    new BufferedReader(new
InputStreamReader(urlConnection.getInputStream()));
            String inputLine;


            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }



            // MAKE SURE TO CLOSE YOUR CONNECTION!
            in.close();




        }
        catch (FileNotFoundException e) {
            System.out.print("URL not found: ");
            System.out.println(e.getMessage());
        }
        catch (MalformedURLException e) {
            System.out.print("Malformed URL: ");
            System.out.println(e.getMessage());
        }
        catch (ProtocolException e) {
            System.out.print("Unsupported protocol: ");
            System.out.println(e.getMessage());
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }

        return response.toString();
    }
}
/**
 * @author: Shawn Struble
 */
package model.data_store;

import model.components.Airport;
import model.components.Flight;
import sun.awt.SunHints;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class FlightStore {
    private static String flightsFilePath;
    private static FlightStore instance;
    private Map<Integer,Flight> flights = new HashMap();
    private FlightStore(String filePath) throws IOException, ParseException {
        flightsFilePath = filePath;
        readFlights();

    }
    /**
     * Throws exception if called before FlightStore singleton instance is
instantiated
     * @return singleton instance of FlightStore
     */
    public static FlightStore getInstance() {
        if (instance == null) {
            throw new RuntimeException("FlightStore not properly instantiated!");
        }
        return instance;
    }
    /**
     *
     * @param filePath
     * @return the singleton instance of FlightStore
     */
    public static FlightStore getInstance(String filePath) throws Exception {
        if(instance == null) {
            instance = new FlightStore(filePath);
        }
        return instance;
    }
    public Flight getFlight(Integer flightNum) {
        return flights.get(flightNum);
    }
    /**
     * populate the flights list from file
     *
     * The list of TTA flights between airports in its route network is in
comma-separated-value format with six fields per line
     * as origin-airport-code,destination-airport-code,departure-time,
arrival-time,flight-number,airfare.
     * The departure and arrival times are in the local time for each airport.
     * Some flights arrive on the day after departure, i.e. the departure time is pm
and the arrival time is am in the next day.
     * The AFRS will read this file on startup.
     */
    private void readFlights() throws IOException, ParseException {
        String line;
        AirportStore store = AirportStore.getInstance();
        try(BufferedReader br = new BufferedReader(new FileReader(flightsFilePath)))
{
            while((line = br.readLine()) != null) {
                //Variables to make flight object
                Airport originAP,destinationAP;
                String arrivalTime,departTime;
                int flightNum, flightAirfare;
                String[] flightsInfo = line.split(",");
                //Parse the info
                originAP = store.getAirportObject(flightsInfo[0]);
                destinationAP = store.getAirportObject(flightsInfo[1]);
                departTime = flightsInfo[2];
                arrivalTime = flightsInfo[3];
                flightNum = Integer.parseInt(flightsInfo[4]);
                flightAirfare = Integer.parseInt(flightsInfo[5]);
                //Create the object and add it to flight list
                Flight flightObj = new
Flight(originAP,destinationAP,departTime,arrivalTime,flightNum,flightAirfare);
                flights.put(flightNum,flightObj);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    /**
     *
     * @return list containing all flights
     */
    public List<Flight> getAllFlights() {
        List<Flight> returnList = new ArrayList<Flight>(flights.values());
        return returnList;
    }

    /**
     *@param orgCode code to match to all flights origin codes
     * @return a list of flights with common origin.
     */
    public List<Flight> getOrgFlights(String orgCode) {
        List<Flight> returnList = new ArrayList<>();
        List<Flight> allFlights = getAllFlights();
        for(int i = 0; i < allFlights.size(); i++ ) {

if(allFlights.get(i).getOrigin().getCode().equals(orgCode.toUpperCase())) {
                returnList.add(allFlights.get(i));
            }
        }
        return returnList;
    }
    /**
     *
     * @param destCode code to match to all flights destination code
     * @return a list of flights with common destination
     */
    public List<Flight> getDestFlights(String destCode) {
        List<Flight> returnList = new ArrayList<>();
        List<Flight> allFlights = getAllFlights();
        for(int i = 0; i < allFlights.size(); i++ ) {

if(allFlights.get(i).getDestination().getCode().equals(destCode.toUpperCase())) {
                returnList.add(allFlights.get(i));
            }
        }
        return returnList;
    }
}
/**
 * @author: Shawn Struble
 */
package model.data_store;

import model.components.Airport;
import model.components.Flight;
import model.components.Itinerary;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.concurrent.TimeUnit;

/**
 * TODO: Store most recently searched itineraries.
 *
 */
public class ItineraryStore {
    private HashMap<Integer, Itinerary> itineraryMap = new HashMap<>();
    private static ItineraryStore instance = null;

    /**
     *
     * @return singleton instance of itinerarystore
     */
    public static ItineraryStore getInstance() {
        if(instance == null) {
            try {
                instance = new ItineraryStore();
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
        return instance;
    }
    private ItineraryStore () throws ParseException {

    }
    /**
     *
     * @param orgCode
     * @param destCode
     * @return list of direct flight itineraries
     */
    private List<Itinerary> getDirectItineries(String orgCode, String destCode) {
        FlightStore instance = FlightStore.getInstance();
        List<Flight> allFlights = instance.getAllFlights();
        List<Flight> matchingFlights = new ArrayList<>();

        for(int i = 0; i < allFlights.size(); i++) { //Get all direct flights with 
matching orig and dest
            if(allFlights.get(i).getOrigin().getCode().equals(orgCode.toUpperCase()) 
&& 

allFlights.get(i).getDestination().getCode().equals(destCode.toUpperCase())) {
                matchingFlights.add(allFlights.get(i));
            }
        }

        List<Itinerary> returnItineraries = new ArrayList<>();
        for(int i = 0; i < matchingFlights.size(); i++) {
            List<Flight> flightList = new ArrayList<>();
            flightList.add(matchingFlights.get(i));
            Itinerary newItinerary = new Itinerary(flightList);
            returnItineraries.add(newItinerary);
        }

        return returnItineraries;
    }

    /**
     * 
     * @param orgCode - code for starting airport
     * @param destCode - code for ending airport
     * @return a list of itineraries with just one connection to reach ending 
airport
     */
    private List<Itinerary> getOneConnectionItinerary(String orgCode, String 
destCode) {
        FlightStore instance = FlightStore.getInstance();
        List<Flight> allFlights = instance.getAllFlights();
        List<Flight> matchingOrgFlights = instance.getOrgFlights(orgCode);
        List<Flight> matchingDestFlights = instance.getDestFlights(destCode);
        List<Itinerary> returnItinerary = new ArrayList<>();
        for(int i = 0; i < matchingOrgFlights.size(); i++) {
            for(int j = 0; j < matchingDestFlights.size(); j++) {
                if
(matchingOrgFlights.get(i).getDestination().getCode().equals(matchingDestFlights.get
(j).getOrigin().getCode())) {
                    long diffInMilli = 
matchingDestFlights.get(j).getArrival().getTime() - 
matchingOrgFlights.get(i).getDeparture().getTime();
                    int diffInMinutes = (int) TimeUnit.MINUTES.convert(diffInMilli, 
TimeUnit.MILLISECONDS);

                    if(diffInMinutes >= 
matchingDestFlights.get(j).getOrigin().getDelayTime() + 
matchingDestFlights.get(j).getOrigin().getConnectionTime()) {
                        List<Flight> flightList = new ArrayList<>();
                        flightList.add(matchingOrgFlights.get(i));
                        flightList.add(matchingDestFlights.get(j));
                        Itinerary newItinerary = new Itinerary(flightList);
                        returnItinerary.add(newItinerary);
                    }
                }
            }
        }
        return returnItinerary;


    }

    /**
     *
     * @param orgCode - starting airport code
     * @param destCode - ending airport code
     * @return a list of itineraries with 2 addition legs to reach destination (3 
flights total)
     */
    private List<Itinerary> getTwoConnectionItinerary(String orgCode, String 
destCode) {
        FlightStore instance = FlightStore.getInstance();
        List<Flight> allFlights = instance.getAllFlights();
        List<Flight> matchingOrgFlights = instance.getOrgFlights(orgCode);
        List<Flight> matchingDestFlights = instance.getDestFlights(destCode);
        List<Itinerary> returnItinerary = new ArrayList<>();
        /*
            For all of the flights with matching origin codes, gets destination code 
and
                compares to all other flights origins.
                    when a match is found tests to make sure that connection is 
possible by the required time restraints
                        for the possible connecting flight compares to the list of 
with ending destinations.
                            same process is repeated as above except with connecting 
flight and possible destination flight.
                                When a match is found an itinerary is created and 
added to the list to be returned.
         */
        for(int i = 0; i < matchingOrgFlights.size(); i++) {
            for(int j = 0 ; j <allFlights.size(); j++) {

if((matchingOrgFlights.get(i).getDestination().getCode().equals(allFlights.get(j).ge
tOrigin().getCode())) &&
(!(allFlights.get(j).getOrigin().getCode().equals(destCode.toUpperCase()))))) {
                    long diffInMilli = allFlights.get(j).getArrival().getTime() -
matchingOrgFlights.get(i).getDeparture().getTime();
                    int diffInMinutes = (int) TimeUnit.MINUTES.convert(diffInMilli,
TimeUnit.MILLISECONDS);
                    if(diffInMinutes >= allFlights.get(j).getOrigin().getDelayTime()
+ allFlights.get(j).getOrigin().getConnectionTime()) {
                        for(int z = 0; z < matchingDestFlights.size(); z++) {

if((allFlights.get(j).getDestination().getCode().equals(matchingDestFlights.get(z).g
etOrigin().getCode())) &&
(!(allFlights.get(j).getDestination().getCode().equals(orgCode.toUpperCase()))))) {
                                diffInMilli =
matchingDestFlights.get(z).getArrival().getTime() -
allFlights.get(j).getDeparture().getTime();
                                diffInMinutes =
(int)TimeUnit.MINUTES.convert(diffInMilli, TimeUnit.MILLISECONDS);
                                if(diffInMinutes >
matchingDestFlights.get(z).getOrigin().getDelayTime() +
matchingDestFlights.get(z).getOrigin().getConnectionTime()) {
                                    List<Flight> flightList = new ArrayList<>();
                                    flightList.add(matchingOrgFlights.get(i));
                                    flightList.add(allFlights.get(j));
                                    flightList.add(matchingDestFlights.get(z));
                                    Itinerary newItinerary = new
Itinerary(flightList);
                                    returnItinerary.add(newItinerary);
                                }
                            }
                        }
                    }
                }

            }
        }


        return returnItinerary;
    }
    /**
     * Returns all itineraries with origin/destination combination
     * @param origin - starting airport code
     * @param destination - ending airport code
     * @return
     */
    public List<Itinerary> getItineraries(String origin,String destination) {
        List<Itinerary> returnItinerary = new ArrayList<>();
        List<Itinerary> directs = getDirectItineries(origin,destination);
        List<Itinerary> single = getOneConnectionItinerary(origin,destination);
        List<Itinerary> multi = getTwoConnectionItinerary(origin,destination);
        for (int i = 0; i < directs.size(); i++) {
            returnItinerary.add(directs.get(i));
        }
        for(int i = 0; i < single.size(); i++) {
            returnItinerary.add(single.get(i));
        }

        for(int i = 0; i < multi.size(); i++) {
            returnItinerary.add(multi.get(i));
        }

        return returnItinerary;

    }
    /**
     *
     * @param origin starting airport code
     * @param destination ending airport code
     * @param legs max amount of connections
     * @return list with corresponding origins and destination and max number of
legs
     */
    public List<Itinerary> getItineraries(String origin, String destination, int
legs) {
        if(legs == 0) {
            return getDirectItineries(origin,destination);
        }
        if(legs == 1) {
            return getOneConnectionItinerary(origin, destination);
        }
        else {
            return getTwoConnectionItinerary(origin,destination);
        }

    }

    public enum FilterBy {
        ORIGIN,
        DESTINATION
    }
    /**
```

```
     * Filter a list of itineraries by their origin or destination airport.
     *
     * @param itineraries original list of itineraries to be filtered
     * @param airportCode airport to use in filter, must be valid
     * @param filterBy is the airport the origin or destination
     * @return filtered list of itineraries
     */
    public static List<Itinerary> filterItins(List<Itinerary> itineraries, String
airportCode, FilterBy filterBy) {
        AirportStore airportStore = AirportStore.getInstance();
        assert airportStore.isAirport(airportCode);
        Airport airport = airportStore.getAirportObject(airportCode);

        List<Itinerary> filteredItins = new ArrayList<>();

        if (filterBy == FilterBy.ORIGIN) {
            for (Itinerary itinerary : itineraries) {
                if (itinerary.getOrigin().equals(airport)) {
                    filteredItins.add(itinerary);
                }
            }
        } else {
            for (Itinerary itinerary : itineraries) {
                if (itinerary.getDestination().equals(airport)) {
                    filteredItins.add(itinerary);
                }
            }
        }

        return filteredItins;
    }
    /**
     * Creates a new itinerary
     * @param flightNums - correspond to the flight objects to put into the
itinerary
     * @return a new itinerary composed of flight objects corresponding to the
parameter with the flight numbers
     */
    public Itinerary createItin(List<Integer> flightNums) {
        FlightStore instance = FlightStore.getInstance();
        List<Flight> flightList = new ArrayList<>();
        for(int i = 0; i < flightNums.size(); i++) {
            flightList.add(instance.getFlight(flightNums.get(i)));
        }
        Itinerary returnItinerary = new Itinerary(flightList);
        return returnItinerary;
    }
}
```

```
/**
 * @author: Shawn Struble
 */
package model.data_store;

import model.components.Itinerary;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

/**
 * TODO: Rewrite delete command
 */
public class ReservationStore {

    public enum Result {
        SUCCESS,
        NOT_FOUND,
        ALREADY_RESERVED
    }

    private static String reservationFilePath;
    public static ReservationStore instance;
    private HashMap<String, List<Itinerary>> reservations = new HashMap<>();

    /**
     * @param filepath file to persist reservations to
     * @throws IOException
     */
    private ReservationStore(String filepath) throws IOException {
        reservationFilePath = filepath;
        readReservations();
        //readReservations();
    }

    public static ReservationStore getInstance(String filepath) throws IOException {
        if (instance == null) {
            instance = new ReservationStore(filepath);
        }
        return instance;
    }

    public static ReservationStore getInstance() {
        if (instance == null) {
            throw new RuntimeException("ReservationStore not properly
instantiated!");
```

```
        }
        return instance;
    }

    /**
     * adds a reservation for a passenger with an itinerary.
     *
     * @param passengerName
     * @param passItinerary
     * @return an enum confirming or denying a reservation
     */
    public Result reserve(String passengerName, Itinerary passItinerary) {
        /*
            Check to see if passenger already has reservation(s)
            Check to ensure cannot make reservations for same origin/destination
combination
         */
        ItineraryStore itineraryStore = ItineraryStore.getInstance();

        if (reservations.containsKey(passengerName)) {
            if (reservations.get(passengerName).contains(passItinerary)) {
                return Result.ALREADY_RESERVED;
            }
            reservations.get(passengerName).add(passItinerary);
            save();
            return Result.SUCCESS;

        }
        /*
            Case for new passenger
            ->Create New Reservation
            ->Create new list as value, and passenger name as key
        */
        List<Itinerary> itineraryList = new ArrayList<>();
        itineraryList.add(passItinerary);
        reservations.put(passengerName, itineraryList);
        save();
        return Result.SUCCESS;
    }

    /**
     * removes a reservation for a passenger's specific itinerary
     * checks to see if passenger has any reservations, then checks if the codes are
registered with them
     * removes passenger key from list if they have no itineraries associated with
them
     *
     * @param passengerName name of the passenger to delete for
     * @param originCode    the three letter code of the origin airport for the 
itinerary
     * @param destCode      the three letter code of the destination airport for the 
itinerary
     * @return null if no itinerary found or the itinerary removed
     */
    public Itinerary delete(String passengerName, String originCode, String 
destCode) {

        if (!(reservations.containsKey(passengerName))) {
            return null;
        }
        List<Itinerary> possibleItineraries = reservations.get(passengerName);
        for (int i = 0; i < possibleItineraries.size(); i++) {
            String origin = possibleItineraries.get(i).getOrigin().getCode();
            String destination = 
possibleItineraries.get(i).getDestination().getCode();
            if (originCode.toUpperCase().equals(origin) && 
destCode.toUpperCase().equals(destination)) { //Found the itinerary
                Itinerary removed = possibleItineraries.remove(i); // Remove the 
itinerary

                if (reservations.get(passengerName).size() == 0) {  //Remove 
passenger if they have no itineraries left
                    reservations.remove(passengerName);

                }
                save();
                return removed;
            }
        }
        return null;


    }

    /**
     * Delete reservation based on given itinerary.
     */
    public Result delete(String passengerName, Itinerary itinerary) {
        if (!(reservations.containsKey(passengerName))) {
            return Result.NOT_FOUND;
        } else if (!reservations.get(passengerName).contains(itinerary)) {
            return Result.NOT_FOUND;
        }

        reservations.get(passengerName).remove(itinerary);
        return Result.SUCCESS;
    }


    /**
     * Returns a passenger's itinerary list
     *
     * @param passengerName the name of the person whos itinerary list you want
     * @return list of Itineraries which the passenger had reserved, this will be
empty if there were none
     */
    public List<Itinerary> retrieve(String passengerName) {
        if (reservations.containsKey(passengerName)) {
            return reservations.get(passengerName);
        }
        return new ArrayList<>();
    }

    /**
     * Returns a passenger's reservation list.
     *
     * @param passengerName the name of the person whos itinerary list you want
     * @param origin        origin airport, may be empty string (will ignore), must
be valid airport
     * @param destination   destination airport, may be empty string (will ignore),
must be valid airport
     * @return list of Itineraries which the passenger had reserved
     */
    public List<Itinerary> retrieve(String passengerName, String origin, String
destination) {
        List<Itinerary> itineraries = retrieve(passengerName);

        if (!origin.equals("")) {
            itineraries = ItineraryStore.filterItins(itineraries, origin,
ItineraryStore.FilterBy.ORIGIN);
        }

        if (!destination.equals("")) {
            itineraries = ItineraryStore.filterItins(itineraries, destination,
ItineraryStore.FilterBy.DESTINATION);
        }

        return itineraries;
    }

    /**
     * read reservations from file into memory
     */
    private void readReservations() throws IOException {
        String line = "";
        BufferedReader br = new BufferedReader(new FileReader(reservationFilePath));
        try {
            while ((line = br.readLine()) != null) {
                List<Integer> flightIds = new ArrayList<>();
                String[] res = line.split("/");
                String passengerName = res[0];
                for (int i = 1; i < res.length; i++) {
                    String[] flightInts = res[i].split(",");
                    List<Integer> FlightNums = new ArrayList<>();

                    for(int j = 0; j < flightInts.length; j++) {
                        FlightNums.add(Integer.parseInt(flightInts[j]));

                    }
                    ItineraryStore instance = ItineraryStore.getInstance();
                    Itinerary newItinerary = instance.createItin(FlightNums);
                    reserve(res[0],newItinerary);

                }


            }
            br.close();


        } catch (IOException e) {
            e.printStackTrace();
        }


    }


    /**
     *
     */
    private void save() {
        BufferedReader br = null;
        String[] keys = reservations.keySet().toArray(new
String[reservations.size()]);

        try {
            //Set up writing to file
            File file = new File(reservationFilePath);
            FileWriter writer = new FileWriter(file);
            BufferedWriter bw = new BufferedWriter(writer);
            //Loop through each key pair
            for (int i = 0; i < reservations.size(); i++) {
                StringBuilder writeString = new StringBuilder();
                List<Itinerary> itineraries = reservations.get(keys[i]);
                writeString.append(keys[i]);

                for (int j = 0; j < itineraries.size(); j++) {
                    writeString.append("/");
                    List<Integer> flightNums = 
itineraries.get(j).getFlightIntegers();
                    for (int z = 0; z < flightNums.size(); z++) {
                        writeString.append(String.valueOf(flightNums.get(z)) + ",");

                    }
                }
                bw.write(String.valueOf(writeString));
                bw.newLine();

            }
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
package model.data_store;

import control.request_response.ClientIdentifierObserver;
import control.request_response.ClientIdentifierSubject;

import java.util.HashMap;


public class ServerStore implements ClientIdentifierObserver {

    private static ServerStore instance = null;

    public enum Server {
        local,faa
    }
    private HashMap<Integer,Server> clientServer = new HashMap<>();
    private ServerStore(ClientIdentifierSubject IDmanager) {
        IDmanager.attachObserver(this);
    }

    public static ServerStore getInstance(ClientIdentifierSubject IDmanager) {
        if(instance == null) {
            instance = new ServerStore(IDmanager);
        }
        return instance;
    }

    public static ServerStore getInstance() {
        if(instance == null) {
            throw new RuntimeException("Server store not properly instantiated");
        }
        return instance;
    }

    public void changeServer(int id,Server newServer) {
        if(clientServer.containsKey(id)) {
            clientServer.put(id,newServer);
        }
    }

    public ServerStore.Server getServer(int id) {
        if(clientServer.containsKey(id)) {
            return clientServer.get(id);
        }
        return null;
    }

    /**
     * Notify observer that a new client identification number has been assigned.
     *
     * @param clientID the new client identification number
     */
    @Override
    public void notifyNewClient(int clientID) {
        clientServer.put(clientID, Server.local);
    }

    /**
     * Notify observer that a client has disconnected.
     *
     * @param clientID the identification number of the client that disconnected
     */
    @Override
    public void notifyClientDisconnect(int clientID) {
        if(clientServer.containsKey(clientID)) {
            clientServer.remove(clientID);
        }
    }
}
/**
```

```
 * CLIClient class. Handles the user input and output interactions,
 * and composes the PTUI.
 * Takes input from console, passes it off to the RequestHandler,
 * and outputs the returned string. Continues until terminated by user.
 *
 * @author Tom Amaral
 */

package view.client;

import control.request_response.RequestHandler;

import java.util.Scanner;

public class CLIClient {


    private final RequestHandler handler;

    public CLIClient(RequestHandler rh){
        handler = rh;
    }


    /**
     * Main outer loop for the AFRS system. Continues to take input from the user
     * until the keyword 'quit' is given. Also outputs a help screen when requested.
     */
    public void run(){

        header();

        String input;

        Scanner in = new Scanner(System.in);

        while(true) {
            System.out.print(">>:");

            input = in.nextLine();

            if(input.equals("quit")){
                System.out.println("Thank you for using AFRS. Have a nice day!");
                break;
            }
            else if(input.equals("help")){
                help();
                continue;
            }

            System.out.println(handler.makeRequest(input));


            System.out.println();
        }
    }


    private void header(){
        System.out.println("--------------------------------------------");
        System.out.println("-----         Tree Top Airways          -----");
        System.out.println("----- Airline Flight Reservation Server -----");
        System.out.println("--------------------------------------------");
        System.out.println();
        System.out.println("Thank you for using TTA's AFRS.");
        System.out.println();
        System.out.println("Please enter a command.");
        System.out.println("'help' for a list, 'quit' to terminate.");
        System.out.println();
    }

    private void help(){
        System.out.println("--------------------------------------------");
        System.out.println("--- AFRS r1 Strategic Slackers March 2018 ---");
        System.out.println("--------------------------------------------");
        System.out.println("Commands:");
        System.out.println("info, reserve, retrieve, delete, airport");
        System.out.println("format: command,paramter(s);");
        System.out.println("Please separate all command parameters with a comma,");
        System.out.println("and terminate commands with a semi-colon.");
        System.out.println("Parameters in brackets are optional.");
        System.out.println();
        System.out.println("info,origin,destination[,connections[,sort-order]];");
        System.out.println();
        System.out.println("reserve,id,passenger;");
        System.out.println();
        System.out.println("retrieve,passenger,[,origin[,destination]];");
        System.out.println();
        System.out.println("delete,passenger,origin,destination;");
        System.out.println();
        System.out.println("airport,airport");
        System.out.println("--------------------------------------------");
    }


}
```

```
package view.client;


import control.request_response.RequestHandler;
import javafx.event.ActionEvent;
import javafx.event.Event;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;


/**
 * Customized Tab component. Has all the components needed for a client
 * tab interface preloaded into it's children.
 *
 *
 * @author Tom '<3' Amaral
 * March 28th, 2017
 */

public class ClientTab extends javafx.scene.control.Tab {

    private int clientId;

    private BorderPane grid;

    private TextField inputField;

    private Button submit;

    private TextArea outputArea;

    private String requestString;

    private RequestHandler requestHandler;


    public ClientTab(RequestHandler requestHandler){

        String cid = requestHandler.makeRequest("connect;");
        this.clientId = Integer.parseInt(cid.substring(8));


        this.requestHandler = requestHandler;

        this.setText("Client " + clientId);

        setup();

        this.setContent(grid);

        //notify request handler when the tab is closed
        this.setOnClosed(new EventHandler<Event>() {
            @Override
            public void handle(Event event) {
                requestHandler.makeRequest(clientId+",disconnect;");
            }
        });

    }


    private void setup(){

        //initialize base nodes

        inputField = new TextField();

        submit = new Button();

        outputArea = new TextArea();

        grid = new BorderPane();

        //customize base nodes

        inputField.setPrefColumnCount(30);

        inputField.setOnKeyPressed(new EventHandler<KeyEvent>() {
            @Override
            public void handle(KeyEvent keyEvent) {
                if(keyEvent.getCode().equals(KeyCode.ENTER)){
                    requestString = inputField.getText();

                    String response =
requestHandler.makeRequest(clientId+","+requestString);
```

```
                updateOutput(response);

                inputField.clear();
            }
        }
    });

    submit.setText("Submit");

    submit.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent actionEvent) {
            requestString = inputField.getText();

            String response = 
requestHandler.makeRequest(clientId+","+requestString);

            updateOutput(response);

            inputField.clear();
        }
    });

    outputArea.setEditable(false);
    outputArea.setMinWidth(740);

    //add base nodes to respective containers

    HBox requestBox = new HBox(inputField, submit);
    HBox output = new HBox(outputArea);

    //customize containers

    requestBox.setSpacing(20);
    requestBox.setPadding(new Insets(10));

    output.setPadding(new Insets(10));

    output.setMaxHeight(350);
    output.setMinHeight(350);
    output.setMaxWidth(700);
    output.setMinWidth(700);

    grid.setPadding(new Insets(10,10,10,10));
    grid.setStyle("-fx-background-color: white");
    grid.setTop(requestBox);
    grid.setBottom(output);
    }


    public void updateOutput(String output){

        outputArea.appendText(output + System.getProperty("line.separator") +
System.getProperty("line.separator"));

    }



}
/**
 * Main front end class for the AFRS system.
 *
 * @author Tom '<3' Amaral
 */


package view.client;

import com.sun.xml.internal.bind.v2.model.core.ID;
import control.request_response.ClientIdentifierManager;
import control.request_response.CommandFactory;
import control.request_response.RequestHandler;
import control.request_response.UndoManager;
import control.request_response.commands.InfoQueryStore;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TabPane;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import model.data_store.*;

public class FrontEnd extends Application {



    private static RequestHandler requestHandler;
    private static ClientIdentifierManager IDmanager;

    public static void main(String[] args) throws Exception{
         IDmanager = new ClientIdentifierManager();

        try {
            AirportStore.getInstance(args[0], args[1], args[2], args[3], IDmanager);
            FlightStore.getInstance(args[4]);
            ReservationStore.getInstance(args[5]);
            ItineraryStore.getInstance();
        } catch (IndexOutOfBoundsException e) {
            System.err.println("INVALID ARGUMENTS! Required: airport file path, 
weather file path, connections file " +
                    "path, delays file path, flights file path, reservations file 
path");
            System.exit(1);
        }

        ServerStore.getInstance(IDmanager);
        InfoQueryStore.getInstance(IDmanager);
        UndoManager undoManager = new UndoManager(IDmanager);
        requestHandler = new RequestHandler(new CommandFactory(), IDmanager, 
undoManager);

        launch(args);
    }


    @Override
    public void start(Stage primaryStage) {

        MainLayoutPane layout = new MainLayoutPane(requestHandler);

        //lights camera action

        primaryStage.setTitle("Airline Flight Reservation System");

        Scene scene = new Scene(layout, 800, 500);

        scene.setFill(Color.rgb(179,0,0, .7));

        primaryStage.setScene(scene);

        primaryStage.setResizable(false);

        primaryStage.show();
    }
```

```
}
/**
 * Customized Tab node that contains a premade 'Help' dialog
 * to be added to the main TabPane
 *
 * @author Tom '<3' Amaral
 * March 29th, 2018
 */


package view.client;

import javafx.geometry.Insets;
import javafx.scene.control.Tab;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;

public class HelpTab extends Tab {

    public HelpTab(){
        this.setText("Welcome");
        this.setClosable(false);



        Text helpText = new Text();

        StringBuffer hText = new StringBuffer();

        hText.append("Thank you for using Treetop Airway's AFRS!" +
System.getProperty("line.separator"));
        hText.append("To get started, add a new Client tab with the button above." +
System.getProperty("line.separator") + System.getProperty("line.separator"));
        hText.append("The available commands are as follows: " +
System.getProperty("line.separator"));
        hText.append("info, reserve, retrieve, delete, airport, undo, redo, and
server." + System.getProperty("line.separator") +
System.getProperty("line.separator"));

        helpText.setText(hText.toString());

        HBox box = new HBox(helpText);

        box.setPadding(new Insets(10));
        box.setStyle("-fx-background-color: white");

        this.setContent(box);

    }

}
/**
 * Precustomized BorderPane representing the top level of the nodes
 * of the main scene. Maintains the latest count of clients.
 * @author Tom '<3' Amaral
 */

package view.client;

import control.request_response.RequestHandler;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.TabPane;
import javafx.scene.layout.BorderPane;

public class MainLayoutPane extends BorderPane {



    public MainLayoutPane(RequestHandler requestHandler){


        TabPane clientTabs = new TabPane();

        Button newClient = new Button("New Client");

        clientTabs.setMinWidth(780);
        clientTabs.setMaxWidth(780);
        clientTabs.setPadding(new Insets(10,0,0,0));
        clientTabs.setStyle("-fx-alignment: center");
        clientTabs.getTabs().add(new HelpTab());

        newClient.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent actionEvent) {
                ClientTab clientTab = new ClientTab(requestHandler);

                clientTabs.getTabs().add(clientTab);
            }
        });
        this.setTop(newClient);
        this.setCenter(clientTabs);

        this.setPadding(new Insets(10));
        this.setStyle("-fx-background-color: grey");

    }



}
import control.request_response.ClientIdentifierManager;
import control.request_response.CommandFactory;
import control.request_response.RequestHandler;
import control.request_response.UndoManager;
import control.request_response.commands.InfoQueryStore;
import model.data_store.*;
import view.client.CLIClient;

/**
 * Entry point for the AFRS PTUI. Starts up the system and gives the API request 
handler to the client, then launches
 * the client PTUI.
 */
public class RunPTUI {

    /**
     * Command line entry point.
     *
     * @param args airport file path, weather file path, connections file path, 
delays file path, flights file path,
     *             reservations file path
     */
    public static void main(String[] args) throws Exception {
        ClientIdentifierManager IDmanager = new ClientIdentifierManager();

        try {
            AirportStore.getInstance(args[0], args[1], args[2], args[3], IDmanager);
            FlightStore.getInstance(args[4]);
            ReservationStore.getInstance(args[5]);
            ItineraryStore.getInstance();
        } catch (IndexOutOfBoundsException e) {
            System.err.println("INVALID ARGUMENTS! Required: airport file path, 
weather file path, connections file " +
                    "path, delays file path, flights file path, reservations file 
path");
            System.exit(1);
        }

        ServerStore.getInstance(IDmanager);
        UndoManager undoManager = new UndoManager(IDmanager);
        RequestHandler requestHandler = new RequestHandler(new CommandFactory(), 
IDmanager, undoManager);
        InfoQueryStore.getInstance(IDmanager);
        CLIClient CLIClient = new CLIClient(requestHandler);
        CLIClient.run();
    }

}
```