```java
import Controller.Server.AFRS;
import Controller.Server.ProxyServer;

/**
 * This class handles the starting of the server
 */
public class ServerMain {

    /**
     * The main method to run the server
     *
     * @param args the arguments from the command line
     */
    public static void main(String[] args) {
        System.out.println("Starting server");
        AFRS afrs = new AFRS();
        ProxyServer proxy = new ProxyServer(afrs);
        proxy.run();
        System.out.println("Server is running");
    }
}

package View;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

import java.util.Observable;
import java.util.Observer;

/**
 * Creates the GUI for the user to interact with the program
 *
 * @author Amber Harding
 */

public class GUI extends Application implements Observer {

    private Client client;
    private VBox textExchanges = new VBox();
    private Insets insets;
    private Stage stage;


    /**
     * Generates the application window*/
    @Override
    public void start(Stage primaryStage) {
        this.stage = primaryStage;
        this.client = new Client();
        client.addObserver(this);
        client.sendCommand("connect;");

        stage.setTitle("Airline Flight Reservation Server");

        stage.setScene(createScene());
        stage.show();
    }

    private Scene createScene() {
        BorderPane mainPane = new BorderPane();
        mainPane.setPrefHeight(500.0D);
        mainPane.setPrefWidth(500.0D);
        insets = new Insets(30);
        mainPane.setPadding(insets);

        mainPane.setTop(top());
        mainPane.setCenter(center());
        mainPane.setBottom(bottom());

        return new Scene(mainPane);
    }

    /**
     * Method to generate a background
     */
    private Background getBackground(int red, int green, int blue){
        Color color = Color.rgb(red, green, blue);
        return new Background(new BackgroundFill(color,CornerRadii.EMPTY,Insets.EMPTY));
    }

    /**
     * Creates HBox for the top portion of the main pane.
     * Contains event handler for the disconnect button.
     */
    private HBox top() {
        Button connect = new Button("Connect");
        connect.setFont(Font.font("Verdana",20));

        connect.setOnAction(event -> {
            try {
                Stage stage = new Stage();
                GUI newgui = new GUI();
                newgui.start(stage);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        Button disconnect = new Button("Disconnect");
        disconnect.setFont(Font.font("Verdana",20));

        disconnect.setOnAction(event -> stop());

        HBox hBox = new HBox();
        hBox.getChildren().addAll(connect,disconnect);
        hBox.setAlignment(Pos.CENTER);
        hBox.setPadding(insets);
        hBox.setSpacing(10);
        hBox.setBorder(new Border(new BorderStroke(Color.BLACK,
                BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
        hBox.setBackground(getBackground(155,200,255));

        return hBox;
    }

    /**
     * Creates VBox for the top portion of the main pane.
     * Contains event handler for the submit button.
     */
    private VBox center() {
        VBox vBox = new VBox();
        vBox.setSpacing(10);

        TextField textField = new TextField();
        textField.setPrefSize(100,100);
        Button submit = new Button("Submit");
        submit.setFont(Font.font("Verdana",20));
        submit.setAlignment(Pos.CENTER);
        submit.setDisable(true);

        textField.textProperty().addListener(observable -> submit.setDisable(false));

        textField.setOnKeyPressed(event -> {
            if(event.getCode().equals(KeyCode.ENTER)){
                submit.fire();
            }
        });

        submit.setOnAction(event -> {
            String command = textField.getText();
            client.sendCommand(command);
            textField.clear();
            submit.setDisable(true);
        });

        vBox.getChildren().addAll(textField,submit);

        vBox.setBorder(new Border(new BorderStroke(Color.BLACK,
                BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
        vBox.setAlignment(Pos.CENTER);

        return vBox;
    }

    /**
     * Creates VBox for the top portion of the main pane.
     * This VBox will display the responses from the client.
     */
    private VBox bottom() {

        VBox vBox = new VBox();
        Text text = new Text("Response:");
        text.setFont(Font.font ("Verdana", 15));

        ScrollPane scrollPane = new ScrollPane();

        scrollPane.setBackground(getBackground(255,255,255));
        scrollPane.setPrefSize(120, 120);
        scrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);
        scrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);
        scrollPane.setContent(textExchanges);

        vBox.getChildren().addAll(text,scrollPane);
        vBox.setBorder(new Border(new BorderStroke(Color.BLACK,
                BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
        vBox.setSpacing(10);
        vBox.setAlignment(Pos.CENTER);
        vBox.setBackground(getBackground(155,200,255));

        return vBox;
    }

    /**
     * Helper method for when a new client response is generated.
     */
    private void updateTextExchange(String response){
        Text text = new Text(response);
        textExchanges.getChildren().add(text);
    }

    @Override
    public void stop(){
        client.sendCommand("disconnect;");
        client.disconnect();
        stage.close();
    }

    @Override
    public void update(Observable o, Object arg) {
        String response = (String) arg;
        if(response != null){
            updateTextExchange(response);
        }
    }
}

package View;

import Controller.Server.AFRS;
import Controller.Server.Server;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Observable;
import java.util.Scanner;

/**
 * Handles the interactions with the user
 * and communicates with the server
 *
 * @author Matt Antantis
 * @author Mark Vittozzi
 * @author Amber Harding
 */
public class Client extends Observable {

    private String response;
    private boolean partial;
    private Scanner console;
    private String command;
    private Socket socket;
    private ObjectOutputStream out;
    private ObjectInputStream netIn;
    private int id;


    /**
     * Constructor for the View.Client
     */
    public Client() {
        this.response = "";
        this.partial = false;
        this.console = new Scanner(System.in);
        this.command = "";
        try {
            socket = new Socket("localhost", 4567);
            out = new ObjectOutputStream(socket.getOutputStream());
            netIn = new ObjectInputStream(socket.getInputStream());
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

    }

    /**
     * Create a new Client a start the exchange process.
     *
     * @param args none
     */
    public static void main(String[] args) {
        Client client = new Client();
        client.run();
    }

    /**
     * Handles asking for a user input
     * and displaying the response
     */
    private void run() {
        while (!socket.isClosed()) {
            response = "";
            makeRequest();
        }
    }

    /**
     * Handles the user input for the program
     */
    private void makeRequest() {
        System.out.print(">");
        String value = console.next();
        sendCommand(value);
    }

    /**
     * Sends a command over the network
     *
     * @param value the command to be sent
     */
    void sendCommand(String value) {

        // Checks if the last command sent was partial
        if (!partial) {
            command = value;

            // Checks if the command is not a connect request or does not have an id
            if (!(command.equals(Server.CONNECT_REQUEST + ';') ||
            Character.isDigit(command.charAt(0)))) {
                command = id + "," + command;
            }
        } else {
            command += value;
        }

        // Splits the command
        String[] splitCommand = command.split(",");

        // Checks if the command was not a connect request
        if (!command.equals(Server.CONNECT_REQUEST + ';')) {
            splitCommand[0] = String.valueOf(id);
        }

        // Writes the split command to the network
        try {
            out.writeUnshared(splitCommand);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

        getResponse();
    }

    /**
     * Sets the response for the most recent command
     **/
    private void getResponse() {

        // Attempts to get a response from the network
        try {
            response = (String) netIn.readUnshared();
        } catch (IOException | ClassNotFoundException e) {
            System.err.println(e.getMessage());
        }

        // Checks if the response was a partial
        if (this.response.equals(id + "," + AFRS.PARTIAL_REQUEST)) {
            System.out.println("Partial Request, Please finish request");
            partial = true;
        } else {
            String[] split = response.split(",");

            // Checks if the response was a connection
            if (split[0].equals("connect")) {
                id = Integer.parseInt(split[1]);

            } else if (split.length > 1) {
                if (split[1].equals(Server.DISCONNECT_REQUEST)) {
                    disconnect();
                }
            }
            partial = false;
            System.out.println(response);

        }

        super.setChanged();
        super.notifyObservers(response);
    }

    /**
     * Disconnects the client from the network
     */
    void disconnect() {
        try {
            out.close();
            netIn.close();
            socket.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

}

package Controller.Requests;

import Model.Managers.AirportManager;

/**
 * This class performs all actions for finding an airports info
 *
 * @author Amber Harding
 * @author Mark Vittozzi
 */
public class AirportInfoRequest implements Request {

    private String airportCode;
    private AirportManager airportManager;
    private int id;

    /**
     * The AirportInfoRequest Constructor
     *
     * @param command        a string array representing the information from the user
     * @param airportManager the airport manager to be used
     */
    public AirportInfoRequest(String[] command, AirportManager airportManager) {
        this.airportCode = command[2];
        this.id = Integer.parseInt(command[0]);
        this.airportManager = airportManager;
    }

    /**
     * gets required airport information from AirportManager
     *
     * @return info: information to display about intended airport.
     */
    @Override
    public String execute() {
        return airportManager.getAirportInfo(airportCode, id);
    }


}

package Controller.Requests;

import Model.Components.Itinerary;
import Model.Components.Passenger;
import Model.Managers.ReservationManager;

/**
 * This class performs all actions for deleting a reservation
 *
 * @author Amber Harding
 * @author Mark Vittozzi
 * @author Ian Randman
 */
public class DeleteReservationRequest implements UndoableRequest {
    private final static String DELETE_SUCCESSFUL = "delete successful";
    private final static String DELETE_ERROR = "error reservation not found";

    private Passenger passenger;
    private ReservationManager reservationManager;
    private String origin;
    private String destination;
    private Itinerary itinerary;

    /**
     * The DeleteReservationRequest Constructor
     *
     * @param command            a string array representing the information from the
     user
     * @param reservationManager the reservation manager to be used
     */
    public DeleteReservationRequest(String[] command, ReservationManager
    reservationManager) {
        this.passenger = new Passenger(command[2]);
        this.reservationManager = reservationManager;
        this.origin = command[3];
        this.destination = command[4];
    }

    /**
     * Has the reservation manager delete intended reservation
     *
     * @return a string indicating if the reservation deletion was successful
     */
    @Override
    public String execute() {
        itinerary = reservationManager.getReservations(passenger, origin,
        destination).get(0).getItinerary();

        if (reservationManager.removeReservation(passenger, origin, destination))
            return DELETE_SUCCESSFUL;
        else
            return DELETE_ERROR;
    }

```

```java
    /**
     * Has the reservation manager undo the deletion of a reservation
     *
     * @return a message containing the passenger and itinerary
     */
    @Override
    public String undo() {
        reservationManager.addReservation(passenger, itinerary);
        return "delete," + passenger.getName() + "," + itinerary;
    }
}

package Controller.Requests;

import Controller.Sorting.AirfareSort;
import Controller.Sorting.ArrivalSort;
import Controller.Sorting.DepartureSort;
import Controller.Sorting.Sort;
import Model.Components.Airport;
import Model.Components.Itinerary;
import Model.Managers.AirportManager;
import Model.Managers.FlightManager;

import java.util.List;

/**
 * This class performs all actions for obtaining flight information
 *
 * @author Amber Harding
 * @author Matt Antantis
 * @author Mark Vittozzi
 * @author Ian Randman
 * @author Jonathon Chierchio
 */

public class FlightInfoRequest implements Request {

    private static final String ORIGIN_ERROR = "error,unknown origin";
    private static final String DESTINATION_ERROR = "error,unknown destination";
    private static final String CONNECTION_ERROR = "error,invalid connection limit";
    private static final String SORT_ERROR = "error,invalid sort order";

    private static final String DEPARTURE_SORT = "departure";
    private static final String ARRIVAL_SORT = "arrival";
    private static final String AIRFARE_SORT = "airfare";

    private String origin;
    private String destination;
    private int connections;
    private String sortOrder;
    private AirportManager airportManager;
    private FlightManager flightManager;
    private List<Itinerary> requestedItineraries;

    /**
     * The constructor for FlightInfoRequest
     *
     * @param command        a string array representing the information from the user
     * @param airportManager the airport manager needed for this task
     * @param flightManager  the flight manager needed for this task
     */
    public FlightInfoRequest(String[] command, AirportManager airportManager,
    FlightManager flightManager) {

        this.airportManager = airportManager;
        this.flightManager = flightManager;
        this.origin = command[2];
        this.destination = command[3];

        if (command.length > 4 && !command[4].equals("")) {
            try {
                this.connections = Integer.parseInt(command[4]);
            } catch (ArrayIndexOutOfBoundsException e) {
                this.connections = 2;
            }
        } else {
            this.connections = 2;
        }

        if (command.length == 6 && !command[5].equals("")) {
            sortOrder = command[5];
        } else {
            sortOrder = DEPARTURE_SORT;
        }
    }

    /**
     * Receives and returns the flight info from the managers
     *
     * @return the pertinent flight information
     */
    public String execute() {
        Airport originAirport = airportManager.getAirport(origin);
        Airport destinationAirport = airportManager.getAirport(destination);
        StringBuilder resp;

        //Error checking
        if (originAirport == null) {
            //Checks that origin airport exists

            resp = new StringBuilder(ORIGIN_ERROR);

        } else if (destinationAirport == null) {
            //Checks that destination airport exists

            resp = new StringBuilder(DESTINATION_ERROR);

        } else if (connections < 0
                || connections > 2) {
            //Checks connection limit is valid

            resp = new StringBuilder(CONNECTION_ERROR);

        } else if (!isValidSortOrder(sortOrder)) {
            //Checks sort order is valid

            resp = new StringBuilder(SORT_ERROR);

        } else {
            List<Itinerary> itineraries = flightManager.getPotentialItineraries(origin,
            destination, connections);

            resp = new StringBuilder("info," + itineraries.size());
            int count = 1;
            requestedItineraries = itineraries;
            sortItineraries(requestedItineraries);

            for (Itinerary itinerary : requestedItineraries) {
                resp.append("\n").append(count).append(",").append(itinerary.getData());
                count++;
            }
        }

        return resp.toString();

    }

    private boolean isValidSortOrder(String sortOrder) {
        return sortOrder.equals(DEPARTURE_SORT) ||
                sortOrder.equals(ARRIVAL_SORT) ||
                sortOrder.equals(AIRFARE_SORT);
    }

    public List<Itinerary> getRequestedItineraries() {
        return requestedItineraries;
    }

    /**
     * Method used to sort itineraries based on the sortOrder
     *
     * @param itineraries list of itineraries being sorted
     */
    private void sortItineraries(List<Itinerary> itineraries) {
        switch (sortOrder) {
            case ARRIVAL_SORT:
                Sort Arrival = new ArrivalSort();
                Arrival.sort(itineraries);
                break;
            case DEPARTURE_SORT:
                Sort Departure = new DepartureSort();
                Departure.sort(itineraries);
                break;
            case AIRFARE_SORT:
                Sort Airfare = new AirfareSort();
                Airfare.sort(itineraries);
                break;
        }
    }
}

package Controller.Requests;

import Model.Components.Itinerary;
import Model.Components.Passenger;
import Model.Managers.ReservationManager;

import java.util.List;

/**
 * This class performs all actions for making a reservation
 *
 * @author Amber Harding
 * @author Matt Antantis
 * @author Mark Vittozzi
 * @author Ian Randman
 */
public class MakeReservationRequest implements UndoableRequest {

    private static final String RESERVE_SUCCESSFUL = "reserve,successful";
    private static final String DUPLICATE_ERROR = "error, duplicate reservation";
    private static final String INVALID_ID_ERROR = "error, invalid id";

    private int id;
    private Passenger passenger;
    private ReservationManager reservationManager;
    private List<Itinerary> requestedItineraries;
    private String origin;
    private String destination;
    private Itinerary itinerary;

    /**
     * Constructor for MakeReservationRequest
     *
     * @param command               a string array representing the information from the
     user
     * @param reservationManager   The reservation manager needed for this request
     * @param requestedItineraries The list of itineraries that a reservation will be
     selected from
     */
    public MakeReservationRequest(String[] command, ReservationManager
    reservationManager,
                                  List<Itinerary> requestedItineraries) {
        this.reservationManager = reservationManager;
        this.requestedItineraries = requestedItineraries;
        this.id = Integer.parseInt(command[2]);
        this.passenger = new Passenger(command[3]);

    }


    /**
     * Has reservation manager make a reservation from the list of itineraries
     *
     * @return a string that indicates if the reservation was successfully created
     */
    @Override
    public String execute() {
        String info;
        try {
            itinerary = requestedItineraries.get(id - 1);
            origin = itinerary.getOrigin();
            destination = itinerary.getDestination();
            boolean status = reservationManager.addReservation(passenger, itinerary);
            if (status)
                info = RESERVE_SUCCESSFUL;
            else
                info = DUPLICATE_ERROR;


        } catch (IndexOutOfBoundsException e) {
            info = INVALID_ID_ERROR;
        }
        return info;
    }

    /**
     * Has the reservation manager undo the making of a reservation
     *
     * @return a message containing the passenger and itinerary unreserved
     */
    @Override
    public String undo() {
        reservationManager.removeReservation(passenger, origin, destination);

        return "make," + passenger.getName() + "," + itinerary;

    }
}

package Controller.Requests;

/**
 * @author Mark Vittozzi
 * @author Amber Harding
 */
public interface Request {
    String execute();
}

package Controller.Requests;

import Model.Components.Passenger;
import Model.Components.Reservation;
import Model.Managers.ReservationManager;

import java.util.List;
```

```java

/**
 * This class is a command that will retrieve a reservation
 *
 * @author Amber Harding
 * @author Mark Vittozzi
 * @author Ian Randman
 */

public class RetrieveReservationRequest implements Request {

    private Passenger passenger;
    private ReservationManager reservationManager;
    private String origin;
    private String destination;

    /**
     * The RetrieveReservationRequest constructor
     *
     * @param command the command to be processed
     */
    public RetrieveReservationRequest(String[] command, ReservationManager
    reservationManager) {
        this.passenger = new Passenger(command[2]);
        this.reservationManager = reservationManager;
        if (command.length == 4) {
            try {
                this.origin = "";
                this.destination = command[3];
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println(e.getMessage());

            }
        } else if (command.length == 5) {
            this.origin = command[3];
            this.destination = command[4];
        } else {
            this.origin = "";
            this.destination = "";
        }
    }

    @Override
    public String execute() {

        List<Reservation> reservations = reservationManager.getReservations(passenger,
        origin, destination);

        StringBuilder info = new StringBuilder("retrieve," + reservations.size());

        for (Reservation r : reservations) {
            info.append(r.getData());
        }

        return info.toString();
    }


}

package Controller.Requests;

import Model.Managers.AirportManager;

/**
 * This class performs all actions for changing the info server a client is using
 *
 * @author Mark Vittozzi
 */
```

```

public class SetInfoModuleRequest implements Request {


    private String server;
    private AirportManager airportManager;
    private int id;

    /**
     * The AirportInfoRequest Constructor
     *
     * @param command        a string array representing the information from the user
     * @param airportManager the airport manager to be used
     */
    public SetInfoModuleRequest(String[] command, AirportManager airportManager) {
        this.server = command[2];
        this.id = Integer.parseInt(command[0]);
        this.airportManager = airportManager;
    }

    /**
     * Calls the switch client method on its airport manager and returns a string
     * alerting the client that the server was successfully changed
     */
    @Override
    public String execute() {
        airportManager.switchClientModule(id, server);
        return "Server,Successful";

    }
}

package Controller.Requests;

/**
 * Interface for requests that can be undone
 *
 * @author Ian Randman
 */
public interface UndoableRequest extends Request {
    String undo();
}

package Controller.Requests;

import Controller.Server.Server;

import java.util.Stack;

/**
 * @author Ian Randman
 * @author Matt Antantis
 */
public class UndoRedoRequestHandler {

    private final static String ERROR_MESSAGE = "error,no request available";

    private Stack<UndoableRequest> undoStack;
    private Stack<UndoableRequest> redoStack;

    /**
     * The constructor for the UndoRedoRequestHandler
     */
    public UndoRedoRequestHandler() {
        this.undoStack = new Stack<>();
        this.redoStack = new Stack<>();
    }

    /**
```

```
     * Adds a request to the stack of undoable requests
     *
     * @param request the command to add to the undo stack
     */
    public void addRequest(UndoableRequest request) {
        undoStack.push(request);
    }

    /**
     * Undoes the most recent command in the undo stack
     * and adds it to the redo stack
     *
     * @return a message containing the status of the undo action
     */
    public String undo() {
        if (!undoStack.empty()) {
            UndoableRequest request = undoStack.pop();
            redoStack.push(request);
            return Server.UNDO_REQUEST + "," + request.undo();
        }

        return ERROR_MESSAGE;
    }

    /**
     * Performs the most recent command in the redo stack
     * and adds it to the undo stack
     *
     * @return a message containing the status of the redo action
     */
    public String redo() {
        if (!redoStack.empty()) {
            UndoableRequest request = redoStack.pop();
            undoStack.push(request);
            return Server.REDO_REQUEST + "," + request.execute();
        }

        return ERROR_MESSAGE;
    }
}

package Controller.Server;

import Controller.Requests.*;
import Model.Components.Itinerary;
import Model.Managers.AirportManager;
import Model.Managers.FlightManager;
import Model.Managers.ReservationManager;

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

/**
 * Coordinates requests from the client
 * and finds the necessary information
 *
 * @author Amber Harding
 * @author Mark Vittozzi
 * @author Ian Randman
 * @author Matt Antantis
 */
public class AFRS implements Server {


    private AirportManager airportManager;
    private FlightManager flightManager;
    private ReservationManager reservationManager;

```

```java
    private Map<Integer, List<Itinerary>> itineraryMap;
    private Map<Integer, UndoRedoRequestHandler> undoRedoRequestHandlerMap;

    private int clientIDIndex;


    /**
     * Construction for Controller.Server.AFRS. Initializes Model.Managers
     */
    public AFRS() {
        this.airportManager = new AirportManager();
        this.flightManager = new FlightManager(airportManager);
        this.reservationManager = new ReservationManager();

        this.itineraryMap = new HashMap<>();
        this.undoRedoRequestHandlerMap = new HashMap<>();
        this.clientIDIndex = 1;

    }

    /**
     * Adds a connection to the map of connections
     */
    synchronized int addClient() {
        itineraryMap.put(clientIDIndex, null);
        undoRedoRequestHandlerMap.put(clientIDIndex, new UndoRedoRequestHandler());
        clientIDIndex++;
        return clientIDIndex - 1;
    }

    /**
     * Reads in a string, from this a request in created and executed.
     *
     * @param command a string array that is used to create and execute a request
     */
    public synchronized String runCommand(String[] command) {
        Request request;
        String response = "";
        int index = Integer.parseInt(command[0]);

        switch (command[1]) {
            case INFO_REQUEST:
                //create flight info request
                request = new FlightInfoRequest(command, airportManager, flightManager);
                response = request.execute();
                itineraryMap.put(index, ((FlightInfoRequest)
                request).getRequestedItineraries());
                request = null;
                break;

            case RESERVE_REQUEST:
                //create reservation request
                request = new MakeReservationRequest(command, reservationManager,
                itineraryMap.get(index));

                undoRedoRequestHandlerMap.get(index).addRequest((MakeReservationRequest)
                request);

                break;

            case RETRIEVE_REQUEST:
                //create retrieve reservation request
                request = new RetrieveReservationRequest(command, reservationManager);
                break;

            case DELETE_REQUEST:
                //create delete reservation request
                request = new DeleteReservationRequest(command, reservationManager);

                undoRedoRequestHandlerMap.get(index).addRequest((DeleteReservationRequest
                ) request);
                break;

            case AIRPORT_REQUEST:
                //create airport request
                request = new AirportInfoRequest(command, airportManager);
                break;

            case DISCONNECT_REQUEST:
                itineraryMap.remove(index);
                undoRedoRequestHandlerMap.remove(index);
                System.out.println(index + " disconnected");
                return DISCONNECT_REQUEST;

            case UNDO_REQUEST:
                request = null;
                response = undoRedoRequestHandlerMap.get(index).undo();
                break;

            case REDO_REQUEST:
                request = null;
                response = undoRedoRequestHandlerMap.get(index).redo();
                break;
            case CHANGE_MODULE_REQUEST:
                request = new SetInfoModuleRequest(command, airportManager);
                break;

            default:
                request = null;
                response = "error, Unknown request";
                break;
        }

        if (request != null) {
            response = request.execute();
        }

        return response;
    }
}

package Controller.Server;

import java.io.IOException;
import java.net.ServerSocket;

/**
 * The Proxy for all commands being sent to the server
 *
 * @author Ian Randman
 * @author Matt Antantis
 */
public class ProxyServer implements Server {

    private final static int PORT = 4567;
    private AFRS afrs;
    private ServerSocket serverSocket;

    /**
     * The constructor for the ProxyServer
     *
     * @param afrs the real server to defer commands to
     */
    public ProxyServer(AFRS afrs) {
        this.afrs = afrs;
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

    /**
     * Runs the proxy server
     */
    public void run() {

        // While the client is connected
        while (!serverSocket.isClosed()) {
            try {
                ServerThread temp = new ServerThread(serverSocket.accept(), this);
                temp.start();
                System.out.println("new connection made");
            } catch (IOException e) {
                System.err.println(e.getMessage());
            }
        }
    }


    /**
     * Executes a command from the client
     *
     * @param command the command to be executed
     * @return the response from the command
     */
    @Override
    public synchronized String runCommand(String[] command) {

        String response;
        int index;

        String end = command[command.length - 1];

        if (!command[0].contains(CONNECT_REQUEST) ||
        command[0].contains(DISCONNECT_REQUEST)) {
            index = Integer.parseInt(command[0]);
        } else {
            index = 0;
        }

        // Test if command is not complete
        if (end.charAt(end.length() - 1) != ';') {
            return index + "," + PARTIAL_REQUEST;
        } else {
            // Cleans the command for the server
            command = getCorrectCommand(command);

            // Checks if client is not connected and wants to connect
            if (command[0].equals(CONNECT_REQUEST)) {
                index = afrs.addClient();
                response = Server.CONNECT_REQUEST + "," + index;
            }
            // Checks if the user is not connected
            else if (index == 0) {
                response = "error,invalid connection";
            }
            // Checks if the user is connected and wants to connect again
            else if (command[1].equals(CONNECT_REQUEST)) {
                response = "error,connection limit reached";
            }
            // Otherwise, the command can be sent to the server
            else {
                response = index + "," + afrs.runCommand(command);
            }
        }
        return response;
```

```java
    }

    /**
     * Returns command string array without the semicolon at the end
     *
     * @param command a string array containing input
     * @return the same string array with the last char from the last index removed
     */
    private String[] getCorrectCommand(String[] command) {
        String holder = command[command.length - 1];
        command[command.length - 1] = holder.substring(0, holder.length() - 1);
        return command;
    }
}

package Controller.Server;

/**
 * The interface for the proxy and subject servers
 *
 * @author Ian Randman
 * @author Matt Antantis
 * @author Mark Vittozzi
 */
public interface Server {

    String PARTIAL_REQUEST = "partial-request";
    String INFO_REQUEST = "info";
    String RESERVE_REQUEST = "reserve";
    String RETRIEVE_REQUEST = "retrieve";
    String DELETE_REQUEST = "delete";
    String AIRPORT_REQUEST = "airport";
    String CONNECT_REQUEST = "connect";
    String DISCONNECT_REQUEST = "disconnect";
    String UNDO_REQUEST = "undo";
    String REDO_REQUEST = "redo";
    String CHANGE_MODULE_REQUEST = "server";

    String runCommand(String[] command);
}

package Controller.Server;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

/**
 * This class allows the server to always be listening for requests coming in from
 clients. To accomplish this, it is
 * run as a thread.
 */
public class ServerThread extends Thread {
    private Server server;
    private Socket socket;
    private ObjectInputStream netIn;
    private ObjectOutputStream out;

    /**
     * Set up the input and output streams for the client-server connection.
     *
     * @param socket the server's connection to the client
     * @param server the main server
     */
    ServerThread(Socket socket, Server server) {
        this.server = server;
        this.socket = socket;

        try {
            this.netIn = new ObjectInputStream(socket.getInputStream());
            this.out = new ObjectOutputStream(socket.getOutputStream());
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

    /**
     * Alternate between receiving a request from the Client and send a response back
     to the Client.
     */
    @Override
    public void run() {
        try {
            while (!socket.isClosed()) {
                String[] command = (String[]) (netIn.readUnshared());

                String response = server.runCommand(command);

                String[] brokenResponse = response.split(",");
                out.writeUnshared(response);

                if (brokenResponse[1].equals(Server.DISCONNECT_REQUEST)) {
                    out.close();
                    netIn.close();
                    socket.close();
                }

            }
        } catch (IOException | ClassNotFoundException e) {
            System.err.println(e.getMessage());
        }
    }
}

package Controller.Sorting;

import Model.Components.Itinerary;

import java.util.List;

/**
 * Algorithm for sorting the flight itineraries based on airfare
 *
 * @author Jonathon Chierchio
 * @author Ian Randman
 */
public class AirfareSort implements Sort {

    /**
     * method for performing a sort on total airfare of itineraries
     *
     * @param itineraries list of itineraries being sorted
     */
    public void sort(List<Itinerary> itineraries) {

        int length = itineraries.size();

        for (int index = 0; index < length; index++) {
            int pos = index;
            for (int j = index; j < length; j++) {
                if (itineraries.get(j).getAirfare() < itineraries.get(pos).getAirfare())
                    pos = j;
            }

            Itinerary min = itineraries.get(pos);
            itineraries.set(pos, itineraries.get(index));
            itineraries.set(index, min);
```

```java

        }

    }
}

package Controller.Sorting;

import Model.Components.Itinerary;

import java.util.List;

/**
 * Algorithm for sorting the flight itineraries based on arrival time
 *
 * @author Jonathon Chierchio
 * @author Ian Randman
 */
public class ArrivalSort implements Sort {

    /**
     * method for performing a sort on arrival times
     *
     * @param itineraries list of itineraries being sorted
     */
    public void sort(List<Itinerary> itineraries) {

        int length = itineraries.size();

        for (int index = 0; index < length; index++) {
            int pos = index;
            for (int j = index; j < length; j++) {
                if (itineraries.get(j).getArrivalTime() <
                itineraries.get(pos).getArrivalTime())
                    pos = j;
            }

            Itinerary min = itineraries.get(pos);
            itineraries.set(pos, itineraries.get(index));
            itineraries.set(index, min);

        }
    }
}

package Controller.Sorting;

import Model.Components.Itinerary;

import java.util.List;

/**
 * Algorithm for sorting the flight itineraries based on departure time
 *
 * @author Jonathon Chierchio
 * @author Ian Randman
 */
public class DepartureSort implements Sort {

    /**
     * method for performing a sort on departure times
     *
     * @param itineraries list of itineraries being sorted
     */
    public void sort(List<Itinerary> itineraries) {

        int length = itineraries.size();

        for (int index = 0; index < length; index++) {
            int pos = index;
            for (int j = index; j < length; j++) {
                if (itineraries.get(j).getDepartureTime() <
                itineraries.get(pos).getDepartureTime())
                    pos = j;
            }

            Itinerary min = itineraries.get(pos);
            itineraries.set(pos, itineraries.get(index));
            itineraries.set(index, min);

        }
    }
}

package Controller.Sorting;

import Model.Components.Itinerary;

import java.util.List;

/**
 * Deals with the sorting algorithm needed to sort flight itineraries
 *
 * @author Amber Harding
 * @author Ian Randman
 */
public interface Sort {

    void sort(List<Itinerary> itineraries);
}

package Model.Components.Weather;

/**
 * A object to hold information from JSON
 *
 * @author Mark Vittozzi
 */
public class Status implements Comparable {
    private String Type;
    private String Reason;
    private String AvgDelay;
    private String EndTime;
    private String MinDelay;
    private String MaxDelay;

    /**
     * Determines which passed in strings are null
     * Non null strings are added to return statement
     *
     * @return a string representing this delay
     */
    @Override
    public String toString() {

        String type = "";
        String reason = "";
        String avg = "";
        String end = "";
        String minDelay = "";
        String maxDelay = "";

        if (Type != null)
            type = "Type: " + Type + ", ";
        if (Reason != null && !Reason.equals(""))
            reason = "Reason " + reason + ", ";
        if (AvgDelay != null)
            avg = "AvgDelay: " + AvgDelay + ", ";
        if (EndTime != null)
            end = "EndTime: " + EndTime + ", ";
        if (MinDelay != null)
            minDelay = "MinTime: " + MinDelay + ", ";
        if (MaxDelay != null)
            maxDelay = "MaxTime: " + MaxDelay;
        return type + reason + avg + end + minDelay + maxDelay;

    }


    @Override
    public int compareTo(Object o) {
        if (o instanceof Status) {
            return Integer.parseInt(this.MaxDelay) - Integer.parseInt(((Status)
            o).MaxDelay);
        } else
            return 0;
    }
}

package Model.Components.Weather;

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

/**
 * A class for storing and returning weather for an airport
 *
 * @author Mark Vittozzi
 */

public class StoredWeather implements WeatherModule {

    private List<String> storedWeather;
    private String delay;

    public StoredWeather() {
        storedWeather = new ArrayList<>();
        this.delay = "";
    }

    /**
     * @param delay: a string representing the delay at this airport
     */
    public void setDelay(String delay) {
        this.delay = delay;
    }

    /**
     * Adds a string representing weather to the list containing
     * all weathers
     *
     * @param weather: A string representing the weather
     */
    @Override
    public void addWeather(String weather) {
        storedWeather.add(weather);
    }

    /**
     * Checks if the index variable is equal to the size of the weather list
     * this is used for making sure the index variable does not go over
     * the length-1
     *
     * @param index: the index variable used by the object who holds this class
     * @return a boolean representing if the index is equal to the list length
     */
    @Override
    public boolean checkId(int index) {
        return (index == storedWeather.size());
    }

    /**
     * returns the weather string at the specified index
     *
     * @param index the index of the string to be returned
     * @return the weather string
     */
    @Override
    public String getWeather(int index) {
        return storedWeather.get(index) + "," + delay;
    }
}

package Model.Components.Weather;

import java.util.List;
/**
 * An object to hold information from JSON
 *
 * @author Mark Vittozzi
 */

public class Weather {


    List<String> Temp;

    /**
     * @return a string representing the temp list
     */
    public String getTemp(){
        String resp = " Temperature: ";
        for(String t : Temp){
            resp += t + " ";
        }
        return resp;
    }

    /**
     * @return the string representing the temp list
     */
    @Override
    public String toString(){
        return getTemp();
    }


}

package Model.Components.Weather;

/**
 * A interface for the two weather module classes to implement
 * (Stored weather and web weather)
 *
 * @author Mark Vittozzi
 */

public interface WeatherModule {

    String getWeather(int id);

    void addWeather(String weather);

    boolean checkId(int id);

    void setDelay(String delay);
}

package Model.Components.Weather;

import java.util.Collections;
import java.util.List;

/**
 * An object to hold information from JSON
 *
 * @author Mark Vittozzi
 */


public class WebAirport {
    private String Name;
    private int DelayCount;
    private Weather Weather;
    private List<Status> Status;

    /**
     * @return the list of statuses as a single string
     */
    private String getStatus() {
        StringBuilder statusString = new StringBuilder();
        for (int i = 0; i < Status.size(); i++) {
            if (i == Status.size() - 1)
                statusString.append(i + 1).append(" ").append(Status.get(i));
            else
                statusString.append(i + 1).append("
                ").append(Status.get(i)).append("\n");
        }
        return statusString.toString();
    }


    /**
     * @return a string representing the class
     */
    @Override
    public String toString() {
        Collections.sort(Status);
        if (DelayCount > 0)
            return Name + "," + Weather + ",Delay count: " + DelayCount + "\nDelays:
            \n" + getStatus();
        else
            return Name + "," + Weather + ",Delay count: " + DelayCount;
    }
}

package Model.Components.Weather;

import com.google.gson.Gson;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.ProtocolException;
import java.net.URL;

/**
 * A class to gather weather information from the web for an airport
 *
 * @author Mark Vittozzi
 */

```

```
public class WebWeather implements WeatherModule {

    // the URL of the site the weather info is stored
    private static final String URL_PREFACE =
    "https://soa.smext.faa.gov/asws/api/airport/status/";

    private String airportCode;

    /**
     * @param airportCode: Takes in an airport code, this is the airport the weather
     info will be for
     */
    public WebWeather(String airportCode) {
        this.airportCode = airportCode;
    }

    /**
     * @param id an id that is not used but required for interface
     * @return a string representing the weather for an airport
     */
    @Override
    public String getWeather(int id) {
        return generateWeather();
    }

    // A method that is not used but required for interface
    @Override
    public void addWeather(String weather) {
    }

    // A method that is not used but required for interface
    @Override
    public boolean checkId(int id) {
        return false;
    }

    // A method that is not used but is required for interface
    @Override
    public void setDelay(String delay) {

    }

    /**
     * Establishes a connection with the URL and obtains the weather info
     * stores them in objects using GSON
     *
     * @return a string representing the weather
     */
    private String generateWeather() {

        String airline = URL_PREFACE + airportCode;
        WebAirport webAirport = null;

        try {
            URL FAA_URL = new URL(airline);
            HttpURLConnection urlConnection = (HttpURLConnection)
            FAA_URL.openConnection();
            urlConnection.setRequestMethod("GET");
            urlConnection.setConnectTimeout(10000);
            urlConnection.setReadTimeout(10000);
            urlConnection.setRequestProperty("Accept", "application/" + "json");
            BufferedReader in =
                    new BufferedReader(new
                    InputStreamReader(urlConnection.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

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

            webAirport = gson.fromJson(response.toString(), WebAirport.class);
            in.close();

        } 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 webAirport.toString();
    }
}

package Model.Components;

import Model.Managers.WeatherManager;

/**
 * Stores the information for an airport
 *
 * @author Matt Antantis
 * @author Mark Vittozzi
 */
public class Airport {

    private String code;
    private String city;
    private int connectionTime;
    private int delay;
    private WeatherManager weatherManager;


    /**
     * The constructor for the airport object
     *
     * @param code the airport's three letter code
     * @param city the airport's city
     */
    public Airport(String code, String city) {
        this.code = code;
        this.city = city;
        this.weatherManager = new WeatherManager(code);
    }

    /**
     * Sets the connection time for the airport
     *
     * @param connectionTime the time for connections
     */
    public void setConnectionTime(int connectionTime) {
        this.connectionTime = connectionTime;
    }

    /**
     * Adds a weather report to the airport
     *
     * @param weather string containing the weather being added
     */
    public void addWeather(String weather) {
        weatherManager.addWeather(weather);
    }
```

```java

    /**
     * Gets the most up to date weather report for the airport
     *
     * @return up to date weather
     */
    public String getWeather(int id) {
        return weatherManager.getWeather(id);
    }


    /**
     * Sets the delay time for the airport
     *
     * @param delay the time of the delay
     */
    public void setDelay(int delay) {
        this.delay = delay;
        this.weatherManager.setDelay(String.valueOf(delay));
    }

    /**
     * determines overlay at airports
     *
     * @return overlay
     */
    public int getOverlay() {
        return delay + connectionTime;
    }

    /**
     * Determines if an object is equal to this
     *
     * @param o the object being compared
     * @return if the two objects are equal
     */
    public boolean equals(Object o) {
        if (o.getClass().equals(this.getClass())) {
            Airport a = (Airport) o;
            return a.code.equals(this.code);
        }
        return false;
    }

    /**
     * Changes the weatherModule for the user
     *
     * @param clientId an int representing the users ID
     * @param server   the server that the client wants to switch to
     */
    public void changeModule(int clientId, String server) {
        weatherManager.switchModules(clientId, server);
    }
}

package Model.Components;

/**
 * Stores the data for a specific flight
 *
 * @author Matt Antantis
 * @author Ian Randman
 */
public class Flight implements Trip {

    private String originCode;
    private String destinationCode;
    private String departureTime;
    private String arrivalTime;
    private int flightNumber;
    private int airfare;
    private int numericArrival;
    private int numericDeparture;

    /**
     * The constructor for Flight from an array of Strings
     *
     * @param flight the details for the flight
     */
    public Flight(String[] flight) {
        this.originCode = flight[0];
        this.destinationCode = flight[1];
        this.departureTime = flight[2];
        this.arrivalTime = flight[3];
        this.flightNumber = Integer.parseInt(flight[4]);
        this.airfare = Integer.parseInt(flight[5]);
        setNumericTimes();
    }

    /**
     * Tests if the flight is between a requested origin and destination
     *
     * @param origin      the origin airport code
     * @param destination the destination airport code
     * @return if the flight is between the two given airports
     */
    public boolean validFlight(String origin, String destination) {
        return this.originCode.equals(origin) &&
        this.destinationCode.equals(destination);
    }

    /**
     * Gets the airfare cost for the flight
     *
     * @return the airfare for the flight
     */
    @Override
    public int getAirfare() {
        return airfare;
    }

    /**
     * Gets the origin of the trip
     *
     * @return the code for the origin airport
     */
    @Override
    public String getOrigin() {
        return originCode;
    }

    /**
     * Gets the destination of the trip
     *
     * @return the code for the destination airport
     */
    @Override
    public String getDestination() {
        return destinationCode;
    }

    /**
     * converts string time (10:30a) into military time for easier use of sorting
     */
    private void setNumericTimes() {
        if (arrivalTime.charAt(arrivalTime.length() - 1) == 'a')
            numericArrival = 0;
        else
```

```java
            numericArrival = 1200;

        numericArrival += buildTime(arrivalTime);

        if (departureTime.charAt(departureTime.length() - 1) == 'a')
            numericDeparture = 0;
        else
            numericDeparture = 1200;

        numericDeparture += buildTime(departureTime);
    }

    /**
     * helper function for setNumericTimes()
     *
     * @param time time of arrival or departures
     * @return int version of time
     */
    private int buildTime(String time) {
        int num = 0;
        for (int i = 0; i < time.length(); i++) {
            char value = time.charAt(i);
            if (value >= '0' && value <= '9') {
                num += value - 48;
                num *= 10;
            }
        }
        return (num / 10) % 1200;
    }

    /**
     * returns numeric arrival time
     */
    public int getNumericArrival() {
        return numericArrival;
    }

    /**
     * returns numeric departure time
     */
    public int getNumericDeparture() {
        return numericDeparture;
    }

    /**
     * Gets the information to display to the user
     *
     * @return the flight data to show to the user
     */
    public String getData() {
        return flightNumber + "," + originCode + "," + departureTime + "," +
        destinationCode + "," + arrivalTime;
    }

    /**
     * Converts the flight data into a string
     *
     * @return the string version of the flight
     */
    @Override
    public String toString() {
        return originCode + "," + destinationCode + "," + departureTime + "," +
        arrivalTime + "," + flightNumber + "," + airfare;
    }
}

package Model.Components;

/**
```

```
 * This class stores the information for a series of flights
 * between two airports
 *
 * @author Matt Antantis
 * @author Ian Randman
 */
public class Itinerary implements Trip {

    private Flight[] flights;

    /**
     * The constructor for the Itinerary
     *
     * @param flights the list of flights in the itinerary
     */
    public Itinerary(Flight[] flights) {
        this.flights = flights;
    }

    /**
     * Gets the number of flights in the itinerary
     *
     * @return the number of flights
     */
    private int getNumFlights() {
        return flights.length;
    }

    /**
     * Gets the total airfare cost for the trip
     *
     * @return the total airfare for the trip
     */
    @Override
    public int getAirfare() {
        int sum = 0;
        for (Flight f : flights) {
            sum += f.getAirfare();
        }
        return sum;
    }

    /**
     * Gets the data to display to the user
     *
     * @return the data for the itinerary
     */
    public String getData() {
        StringBuilder result = new StringBuilder(getAirfare() + "," + getNumFlights());
        for (Flight f : flights) {
            result.append(",").append(f.getData());
        }
        return result.toString();
    }

    /**
     * Gets the origin of the trip
     *
     * @return the code for the starting airport
     */
    @Override
    public String getOrigin() {
        return flights[0].getOrigin();
    }

    /**
     * Gets the destination of the trip
     *
     * @return the code for the final airport
```

```java
     */
    @Override
    public String getDestination() {
        return flights[flights.length - 1].getDestination();
    }

    /**
     * Converts the itinerary data into a string
     *
     * @return the string version of the itinerary
     */
    @Override
    public String toString() {
        StringBuilder result = new StringBuilder("" + getNumFlights());
        for (Flight f : flights) {
            result.append(",").append(f.toString());
        }
        return result.toString();
    }

    /**
     * obtains the arrival time for the final flight in the itinerary
     *
     * @return integer time of arrival for the itinerary
     */
    public int getArrivalTime() {
        return flights[flights.length - 1].getNumericArrival();
    }

    /**
     * obtains the departure time of an itinerary
     *
     * @return integer time of departure for the itinerary
     */
    public int getDepartureTime() {
        return flights[0].getNumericDeparture();
    }
}

package Model.Components;

/**
 * This class stores data for the Passenger
 * for a reservation
 *
 * @author Matt Antantis
 */
public class Passenger {
    private String name;

    /**
     * The constructor for Passenger
     *
     * @param name the name of the Passenger
     */
    public Passenger(String name) {
        this.name = name;
    }

    /**
     * Gets the name of the passenger
     *
     * @return the passenger's name
     */
    public String getName() {
        return name;
    }

    /**
```

```java
     * Determines if this passenger is equal to another object
     *
     * @param o the object being compared
     * @return if the two objects are equal
     */
    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass())
            return false;

        Passenger that = (Passenger) o;

        return this.name.equals(that.name);
    }
}

package Model.Components;

/**
 * This class stores the reservation for a passenger and an itinerary
 *
 * @author Matt Antantis
 * @author Ian Randman
 */
public class Reservation implements Trip {

    private Itinerary itinerary;
    private Passenger passenger;

    /**
     * The Reservation constructor
     *
     * @param passenger the passenger on the trip
     * @param itinerary the itinerary for the trip
     */
    public Reservation(Passenger passenger, Itinerary itinerary) {
        this.passenger = passenger;
        this.itinerary = itinerary;
    }

    /**
     * Gets the passenger for the trip
     *
     * @return the reservation's passenger
     */
    public Passenger getPassenger() {
        return passenger;
    }

    /**
     * Gets the total airfare cost for the trip
     *
     * @return the total airfare for the trip
     */
    @Override
    public int getAirfare() {
        return itinerary.getAirfare();
    }

    /**
     * Gets the origin of the trip
     *
     * @return the starting airport code
     */
    public String getOrigin() {
        return itinerary.getOrigin();
    }

    /**
```

```
     * Gets the destination of the trip
     *
     * @return the final airport code
     */
    public String getDestination() {
        return itinerary.getDestination();
    }

    public Itinerary getItinerary() {
        return itinerary;
    }

    /**
     * The information from the reservation to be
     * displayed to the user
     *
     * @return the info to be shown to the user
     */
    public String getData() {
        return "\n" + itinerary.getData();
    }

    /**
     * Converts the Reservation data into a string
     *
     * @return the reservation as a string
     */
    @Override
    public String toString() {
        return passenger.getName() + "," + itinerary.toString();
    }

    /**
     * Checks to see if the reservation is equal to an object
     *
     * @param o the object being compared
     * @return if the two objects are equal
     */
    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass())
            return false;

        Reservation that = (Reservation) o;

        if (this.getOrigin().equals(that.getOrigin())) {
            if (this.getDestination().equals(that.getDestination()))
                return passenger.equals(that.passenger);
        }
        return false;
    }

    /**
     * Creates a hash code for the object
     * This method was auto-generated by IntelliJ
     *
     * @return the hashcode of the reservation
     */
    @Override
    public int hashCode() {
        int result = itinerary.hashCode();
        result = 31 * result + passenger.hashCode();
        return result;
    }
}

package Model.Components;

/**
```

```
 * This class is the Component object for the Composite
 * pattern. The methods inside are overridden by its
 * children.
 *
 * @author Matt Antantis
 * @author Ian Randman
 */
public interface Trip {

    /**
     * Gets the airfare cost for a trip
     *
     * @return the airfare cost
     */
    int getAirfare();

    /**
     * Gets the origin of the trip
     *
     * @return the origin of the trip
     */
    String getOrigin();

    /**
     * Gets the destination of the trip
     *
     * @return the destination of the trip
     */
    String getDestination();

    /**
     * Gets the data to show to the user
     *
     * @return the data of the trip
     */
    String getData();
}

package Model.Managers;

import Model.Components.Airport;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

/**
 * This class holds all the airports for the system
 * and responds to requests for information about them.
 *
 * @author Matt Antantis
 * @author Mark Vittozzi
 */
public class AirportManager extends Manager {
    private Map<String, Airport> airports;

    /**
     * The constructor for the AirportManger
     */
    public AirportManager() {
        airports = new HashMap<>();

        try {
            buildList();
            buildConnections();
            buildDelays();
            buildWeather();
```

```java
        } catch (FileNotFoundException e) {
            System.err.println(e.getMessage());
        }
    }

    /**
     * Builds the list of Airports by reading a text file
     *
     * @throws FileNotFoundException throws an exception if cities.txt is not found
     */
    private void buildList() throws FileNotFoundException {
        reader = new Scanner(new FileReader("data/cities.txt"));
        System.out.print("initializing airports");
        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            airports.put(line[0], new Airport(line[0], line[1]));
            System.out.print(".");
        }
        System.out.println("done");

    }

    /**
     * Updates the airports in the list with the connection
     * time form a text file
     *
     * @throws FileNotFoundException if connection_times.txt is not found an exception
     will be thrown
     */
    private void buildConnections() throws FileNotFoundException {
        reader = new Scanner(new FileReader("data/connection_times.txt"));
        System.out.print("initializing connections");
        reader.nextLine();
        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            airports.get(line[0]).setConnectionTime(Integer.parseInt(line[1]));
            System.out.print(".");
        }
        System.out.println("done");
    }

    /**
     * Updates the airports in the list with the delay
     * time from a text file
     *
     * @throws FileNotFoundException if delay_times.txt is not found an exception will
     be thrown
     */
    private void buildDelays() throws FileNotFoundException {
        reader = new Scanner(new FileReader("data/delay_times.txt"));
        System.out.print("initializing delays");
        reader.nextLine();
        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            airports.get(line[0]).setDelay(Integer.parseInt(line[1]));
            System.out.print(".");
        }
        System.out.println("done");
    }

    /**
     * Updates the airports in the list with the weather data
     * from a text file
     *
     * @throws FileNotFoundException if weather.txt is not found
     */
    private void buildWeather() throws FileNotFoundException {
        reader = new Scanner(new FileReader("data/weather.txt"));
        System.out.print("initializing weather");
        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            for (int i = 1; i < line.length; i += 2) {
                airports.get(line[0]).addWeather(line[0] + "," + line[i] + "," + line[i
                + 1]);
            }
            System.out.print(".");
        }
        System.out.println("done");
    }

    /**
     * retrieves airport via airport code
     *
     * @param code code being used to retrieve airport
     * @return airport
     */
    public Airport getAirport(String code) {
        return airports.get(code);
    }

    /**
     * retrieves delay via airport code
     *
     * @param code airport code for airport
     * @return delay time
     */
    int getDelay(String code) {
        return airports.get(code).getOverlay();
    }

    public String getAirportInfo(String airportCode, int id) {
        return airports.get(airportCode).getWeather(id);
    }

    public void switchClientModule(int id, String server) {
        for (String s : airports.keySet()) {
            airports.get(s).changeModule(id, server);
        }
    }

}

package Model.Managers;

import Model.Components.Flight;
import Model.Components.Itinerary;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * This class handles interactions with the list of flights
 *
 * @author Matt Antantis
 */
public class FlightManager extends Manager {

    private ArrayList<Flight> flights;
    private AirportManager airportManager;

    /**
     * The FlightManager constructor
     */
    public FlightManager(AirportManager airportManager) {
        flights = new ArrayList<>();
        this.airportManager = airportManager;
```

```java

        try {
            buildList();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * Builds each flight from a line in a text file
     *
     * @throws FileNotFoundException if flights.txt is not found
     */
    private void buildList() throws FileNotFoundException {
        reader = new Scanner(new FileReader("data/flights.txt"));

        reader.nextLine();
        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            flights.add(new Flight(line));
        }
        System.out.println("Build all flights");
    }

    /**
     * Builds a list of potential flights from an origin and destination
     *
     * @param origin      the origin airport code
     * @param destination the destination airport code
     * @return a list of all potential flights combinations
     */
    public ArrayList<Itinerary> getPotentialItineraries(String origin, String destination, int legs) {
        ArrayList<Flight> potentialFlights = getFlightsBetween(origin, destination);

        // Gather all single leg itineraries
        ArrayList<Itinerary> potentialItineraries = new ArrayList<>();
        for (Flight f : potentialFlights) {
            potentialItineraries.add(new Itinerary(new Flight[]{f}));
        }

        // Checks for possible multi flight itineraries
        if (legs > 0) {
            ArrayList<Flight> fromOrigin = new ArrayList<>();
            ArrayList<Flight> fromDestination = new ArrayList<>();
            for (Flight f : flights) {
                if (f.getOrigin().equals(origin) && !potentialFlights.contains(f)) {
                    fromOrigin.add(f);
                } else if (f.getDestination().equals(destination) && !potentialFlights.contains(f)) {
                    fromDestination.add(f);
                }
            }

            // Checks to see if the two flights form a valid itinerary
            for (Flight ori : fromOrigin) {
                for (Flight desti : fromDestination) {
                    if (checkConnection(ori, desti))
                        potentialItineraries.add(new Itinerary(new Flight[]{ori, desti}));
                }
            }

            // Checks for extra valid itineraries if 2 connections are allowed
            if (legs > 1) {
                ArrayList<Flight> temp;
                for (Flight ori : fromOrigin) {
                    for (Flight desti : fromDestination) {
                        // Gets flights connecting the other two legs
```

```
                            temp = getFlightsBetween(ori.getDestination(),
                            desti.getOrigin());

                            // Checks to see if the three selected flights are valid
                            for (Flight f : temp) {
                                if (checkConnection(ori, f) && checkConnection(f, desti)) {
                                    potentialItineraries.add(new Itinerary(new
                                    Flight[]{ori, f, desti}));
                                }
                            }
                        }
                    }
                }
            }
            return potentialItineraries;
        }

        /**
         * Builds a list of flights between two airports
         *
         * @param origin      the code of the origin airport
         * @param destination the code of the destination airport
         * @return the list of all flights between the two
         */
        private ArrayList<Flight> getFlightsBetween(String origin, String destination) {
            ArrayList<Flight> potential = new ArrayList<>();
            for (Flight f : flights) {
                if (f.validFlight(origin, destination))
                    potential.add(f);
            }
            return potential;
        }

        /**
         * Checks to see if two flights are able to connect
         *
         * @param first  the first flight
         * @param second the connecting flight
         * @return if the two flights can be connected
         */
        private boolean checkConnection(Flight first, Flight second) {
            if (first.getDestination().equals(second.getOrigin())) {
                int overlay = airportManager.getDelay(first.getDestination());
                if (overlay >= 60) {
                    int hours = overlay / 60;
                    overlay = overlay % 60;
                    overlay += hours * 100;
                }
                return overlay + first.getNumericArrival() <= second.getNumericDeparture();
            } else
                return false;
        }

    }

    package Model.Managers;

    import java.io.FileNotFoundException;
    import java.util.Scanner;

    /**
     * Abstract class designed for creating
     * and maintaining a collection of objects
     *
     * @author Matt Antantis
     */
    abstract class Manager {

        Scanner reader;
```

```java

    /**
     * Builds the collection of objects based off the implementation
     *
     * @throws FileNotFoundException if the file with the data is not found
     */
    private void buildList() throws FileNotFoundException {
    }

}

package Model.Managers;

import Model.Components.Flight;
import Model.Components.Itinerary;
import Model.Components.Passenger;
import Model.Components.Reservation;

import java.io.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

/**
 * This class handles all the Reservations for the AFRS
 *
 * @author Matt Antantis
 */
public class ReservationManager extends Manager {

    private ArrayList<Reservation> reservations;

    /**
     * The constructor for the ReservationManager
     */
    public ReservationManager() {
        reservations = new ArrayList<>();

        try {
            buildList();
            System.out.println("..done");
        } catch (FileNotFoundException e) {
            System.out.println("Data not found");

        }
    }

    /**
     * Builds the reservations from a saved text file
     *
     * @throws FileNotFoundException if reservations.txt is not found
     */
    private void buildList() throws FileNotFoundException {
        System.out.print("restoring reservations\t");
        reader = new Scanner(new FileReader("data/reservations.txt"));

        while (reader.hasNext()) {
            String[] line = reader.nextLine().split(",");
            Passenger tempPassenger = new Passenger(line[0]);
            Flight[] tempFlights = new Flight[Integer.parseInt(line[1])];

            int index = 0;
            for (int i = 2; i < line.length; i += 6) {
                tempFlights[index] = (new Flight(Arrays.copyOfRange(line, i, i + 6)));
                index++;
            }

            reservations.add(new Reservation(tempPassenger, new Itinerary(tempFlights)));

        }
    }

    /**
     * Adds a new Reservation to the reservations
     *
     * @param passenger the passenger for the reservation
     * @param itinerary the itinerary for the reservation
     * @return if the reservation was successful
     */
    public boolean addReservation(Passenger passenger, Itinerary itinerary) {

        Reservation temp = new Reservation(passenger, itinerary);

        if (reservations.contains(temp)) {
            return false;
        }
        reservations.add(temp);
        saveData();
        return true;
    }

    /**
     * Removes a reservation from the list
     *
     * @param passenger   the passenger to be removed
     * @param origin      the reservation's starting location
     * @param destination the reservation's ending location
     * @return if the reservation was successfully removed
     */
    public boolean removeReservation(Passenger passenger, String origin, String
    destination) {
        for (Reservation r : reservations) {
            if (passenger.equals(r.getPassenger())) {
                if (origin.equals(r.getOrigin()) &&
                destination.equals(r.getDestination())) {
                    reservations.remove(r);
                    saveData();
                    return true;
                }
            }
        }
        return false;
    }

    /**
     * Gets the reservations for a passenger based on the origin and destination
     *
     * @param passenger   the passenger with reservations
     * @param origin      the passenger's origin
     * @param destination the passenger's destination
     * @return all the passenger's reservations
     */
    public ArrayList<Reservation> getReservations(Passenger passenger, String origin,
    String destination) {
        ArrayList<Reservation> reserved = new ArrayList<>();
        if (origin.equals("")) {
            if (destination.equals("")) {

                for (Reservation r : reservations) {
                    if (passenger.equals(r.getPassenger()))
                        reserved.add(r);
                }
            } else {

                for (Reservation r : reservations) {
                    if (passenger.equals(r.getPassenger()) &&
                    destination.equals(r.getDestination()))
                        reserved.add(r);
                }
            }
        } else if (destination.equals("")) {

            for (Reservation r : reservations) {
                if (passenger.equals(r.getPassenger()) && origin.equals(r.getOrigin()))
                    reserved.add(r);
            }
        } else {

            for (Reservation r : reservations) {
                if (passenger.equals(r.getPassenger())) {
                    if (origin.equals(r.getOrigin()) &&
                    destination.equals(r.getDestination())) {
                        reserved.add(r);

                    }
                }
            }
        }

        return reserved;
    }

    /**
     * Saves the current reservations to a text file
     */
    private void saveData() {
        try {
            BufferedWriter writer = new BufferedWriter(new
            FileWriter("data/reservations.txt"));
            for (Reservation r : reservations) {
                writer.write(r.toString() + "\n");
            }
            writer.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}

package Model.Managers;

import Model.Components.Weather.StoredWeather;
import Model.Components.Weather.WeatherModule;
import Model.Components.Weather.WebWeather;

import java.util.HashMap;

/**
 * An object that creates, holds, interacts with and modifies the weather modules
 * of a specific airport
 *
 * @author Mark Vittozzi
 */

public class WeatherManager extends Manager {


    private HashMap<Integer, WeatherModule> weatherModuleMap;
    private HashMap<Integer, Integer> indexMap;
    private WeatherModule storedWeather;
    private String airportCode;

    /**
     * @param airportCode: A string representing the name of the airport this object
     belongs to
     */
    public WeatherManager(String airportCode) {
        this.weatherModuleMap = new HashMap<>();
        this.indexMap = new HashMap<>();
        this.storedWeather = new StoredWeather();
        this.airportCode = airportCode;
    }

    /**
     * Adds the passed in weather to the module
     *
     * @param weather: A string representing weather
     */
    public void addWeather(String weather) {
        storedWeather.addWeather(weather);
    }

    /**
     * Changes the weather module
     *
     * @param clientId: An int representing the user
     * @param server:   a string representing the server to be switched to
     */
    public void switchModules(int clientId, String server) {
        if (server.equals("local")) {
            weatherModuleMap.replace(clientId, storedWeather);
        } else {
            weatherModuleMap.replace(clientId, new WebWeather(airportCode));
        }
    }

    /**
     * Obtains the weather from the weather module
     * determines if the index variable is too large and resets it
     *
     * @param clientId: An int representing the users id
     * @return a string representing the weather for the airport
     */
    public String getWeather(int clientId) {
        if (!(weatherModuleMap.containsKey(clientId))) {
            indexMap.put(clientId, 0);
            weatherModuleMap.put(clientId, storedWeather);
        }
        if (weatherModuleMap.get(clientId).checkId(indexMap.get(clientId))) {
            indexMap.replace(clientId, 0);
        }

        String weather =
        weatherModuleMap.get(clientId).getWeather(indexMap.get(clientId));
        indexMap.replace(clientId, indexMap.get(clientId) + 1);

        return weather;
    }

    /**
     * sets the delay of the storedWeather Module
     *
     * @param delay a string representing a delay
     */
    public void setDelay(String delay) {
        storedWeather.setDelay(delay);
    }

}
```