

```

1  import Controller.Server.AFRS;
2  import Controller.Server.ProxyServer;
3
4  /**
5   * This class handles the starting of the server
6   */
7  public class ServerMain {
8
9      /**
10     * The main method to run the server
11     *
12     * @param args the arguments from the command line
13     */
14     public static void main(String[] args) {
15         System.out.println("Starting server");
16         AFRS afrs = new AFRS();
17         ProxyServer proxy = new ProxyServer(afrs);
18         proxy.run();
19         System.out.println("Server is running");
20     }
21 }
22
23 package View;
24
25 import javafx.application.Application;
26 import javafx.geometry.Insets;
27 import javafx.geometry.Pos;
28 import javafx.scene.*;
29 import javafx.scene.control.*;
30 import javafx.scene.control.Button;
31 import javafx.scene.control.TextField;
32 import javafx.scene.input.KeyCode;
33 import javafx.scene.layout.*;
34 import javafx.scene.paint.Color;
35 import javafx.scene.text.Font;
36 import javafx.scene.text.Text;
37 import javafx.stage.Stage;
38
39 import java.util.Observable;
40 import java.util.Observer;
41
42 /**
43  * Creates the GUI for the user to interact with the program
44  *
45  * @author Amber Harding
46  */
47
48 public class GUI extends Application implements Observer {
49
50     private Client client;
51     private VBox textExchanges = new VBox();
52     private Insets insets;
53     private Stage stage;
54
55     /**
56     * Generates the application window*/
57     @Override
58     public void start(Stage primaryStage) {
59         this.stage = primaryStage;
60         this.client = new Client();
61         client.addObserver(this);
62         client.sendCommand("connect;");
63
64         stage.setTitle("Airline Flight Reservation Server");
65
66         stage.setScene(createScene());
67         stage.show();
68     }
69 }

```

```

70
71 private Scene createScene() {
72     BorderPane mainPane = new BorderPane();
73     mainPane.setPrefHeight(500.0D);
74     mainPane.setPrefWidth(500.0D);
75     insets = new Insets(30);
76     mainPane.setPadding(insets);
77
78     mainPane.setTop(top());
79     mainPane.setCenter(center());
80     mainPane.setBottom(bottom());
81
82     return new Scene(mainPane);
83 }
84
85 /**
86  * Method to generate a background
87  */
88 private Background getBackground(int red, int green, int blue){
89     Color color = Color.rgb(red, green, blue);
90     return new Background(new BackgroundFill(color, CornerRadii.EMPTY, Insets.EMPTY));
91 }
92
93 /**
94  * Creates HBox for the top portion of the main pane.
95  * Contains event handler for the disconnect button.
96  */
97 private HBox top() {
98     Button connect = new Button("Connect");
99     connect.setFont(Font.font("Verdana", 20));
100
101     connect.setOnAction(event -> {
102         try {
103             Stage stage = new Stage();
104             GUI newgui = new GUI();
105             newgui.start(stage);
106         } catch (Exception e) {
107             e.printStackTrace();
108         }
109     });
110
111     Button disconnect = new Button("Disconnect");
112     disconnect.setFont(Font.font("Verdana", 20));
113
114     disconnect.setOnAction(event -> stop());
115
116     HBox hBox = new HBox();
117     hBox.getChildren().addAll(connect, disconnect);
118     hBox.setAlignment(Pos.CENTER);
119     hBox.setPadding(insets);
120     hBox.setSpacing(10);
121     hBox.setBorder(new Border(new BorderStroke(Color.BLACK,
122         BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
123     hBox.setBackground(getBackground(155, 200, 255));
124
125     return hBox;
126 }
127
128 /**
129  * Creates VBox for the top portion of the main pane.
130  * Contains event handler for the submit button.
131  */
132 private VBox center() {
133     VBox vBox = new VBox();
134     vBox.setSpacing(10);
135
136     TextField textField = new TextField();
137     textField.setPrefSize(100, 100);
138     Button submit = new Button("Submit");

```

```

139         submit.setFont(Font.font("Verdana",20));
140         submit.setAlignment(Pos.CENTER);
141         submit.setDisable(true);
142
143         textField.textProperty().addListener(observable -> submit.setDisable(false));
144
145         textField.setOnKeyPressed(event -> {
146             if(event.getCode().equals(KeyCode.ENTER)){
147                 submit.fire();
148             }
149         });
150
151         submit.setOnAction(event -> {
152             String command = textField.getText();
153             client.sendCommand(command);
154             textField.clear();
155             submit.setDisable(true);
156         });
157
158         vbox.getChildren().addAll(textField,submit);
159
160         vbox.setBorder(new Border(new BorderStroke(Color.BLACK,
161             BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
162         vbox.setAlignment(Pos.CENTER);
163
164         return vbox;
165     }
166
167     /**
168     * Creates VBox for the top portion of the main pane.
169     * This VBox will display the responses from the client.
170     */
171     private VBox bottom() {
172
173         VBox vbox = new VBox();
174         Text text = new Text("Response:");
175         text.setFont(Font.font("Verdana", 15));
176
177         ScrollPane scrollPane = new ScrollPane();
178
179         scrollPane.setBackground(getBackground(255,255,255));
180         scrollPane.setPrefSize(120, 120);
181         scrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);
182         scrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);
183         scrollPane.setContent(textExchanges);
184
185         vbox.getChildren().addAll(text,scrollPane);
186         vbox.setBorder(new Border(new BorderStroke(Color.BLACK,
187             BorderStrokeStyle.SOLID, CornerRadii.EMPTY, BorderWidths.DEFAULT)));
188         vbox.setSpacing(10);
189         vbox.setAlignment(Pos.CENTER);
190         vbox.setBackground(getBackground(155,200,255));
191
192         return vbox;
193     }
194
195     /**
196     * Helper method for when a new client response is generated.
197     */
198     private void updateTextExchange(String response){
199         Text text = new Text(response);
200         textExchanges.getChildren().add(text);
201     }
202
203     @Override
204     public void stop(){
205         client.sendCommand("disconnect;");
206         client.disconnect();
207         stage.close();

```

```

208     }
209
210     @Override
211     public void update(Observable o, Object arg) {
212         String response = (String) arg;
213         if(response != null){
214             updateTextExchange(response);
215         }
216     }
217 }
218
219 package View;
220
221 import Controller.Server.AFRS;
222 import Controller.Server.Server;
223
224 import java.io.IOException;
225 import java.io.ObjectInputStream;
226 import java.io.ObjectOutputStream;
227 import java.net.Socket;
228 import java.util.Observable;
229 import java.util.Scanner;
230
231 /**
232  * Handles the interactions with the user
233  * and communicates with the server
234  *
235  * @author Matt Antantis
236  * @author Mark Vittozzi
237  * @author Amber Harding
238  */
239 public class Client extends Observable {
240
241     private String response;
242     private boolean partial;
243     private Scanner console;
244     private String command;
245     private Socket socket;
246     private ObjectOutputStream out;
247     private ObjectInputStream netIn;
248     private int id;
249
250
251     /**
252      * Constructor for the View.Client
253      */
254     public Client() {
255         this.response = "";
256         this.partial = false;
257         this.console = new Scanner(System.in);
258         this.command = "";
259         try {
260             socket = new Socket("localhost", 4567);
261             out = new ObjectOutputStream(socket.getOutputStream());
262             netIn = new ObjectInputStream(socket.getInputStream());
263         } catch (IOException e) {
264             System.err.println(e.getMessage());
265         }
266     }
267
268
269     /**
270      * Create a new Client a start the exchange process.
271      *
272      * @param args none
273      */
274     public static void main(String[] args) {
275         Client client = new Client();
276         client.run();

```

```

277     }
278
279     /**
280     * Handles asking for a user input
281     * and displaying the response
282     */
283     private void run() {
284         while (!socket.isClosed()) {
285             response = "";
286             makeRequest();
287         }
288     }
289
290     /**
291     * Handles the user input for the program
292     */
293     private void makeRequest() {
294         System.out.print(">");
295         String value = console.next();
296         sendCommand(value);
297     }
298
299     /**
300     * Sends a command over the network
301     *
302     * @param value the command to be sent
303     */
304     void sendCommand(String value) {
305
306         // Checks if the last command sent was partial
307         if (!partial) {
308             command = value;
309
310             // Checks if the command is not a connect request or does not have an id
311             if (!(command.equals(Server.CONNECT_REQUEST + ';') ||
312                 Character.isDigit(command.charAt(0)))) {
313                 command = id + "," + command;
314             }
315             } else {
316                 command += value;
317             }
318
319             // Splits the command
320             String[] splitCommand = command.split(",");
321
322             // Checks if the command was not a connect request
323             if (!command.equals(Server.CONNECT_REQUEST + ';')) {
324                 splitCommand[0] = String.valueOf(id);
325             }
326
327             // Writes the split command to the network
328             try {
329                 out.writeUnshared(splitCommand);
330             } catch (IOException e) {
331                 System.err.println(e.getMessage());
332             }
333
334             getResponse();
335         }
336
337     /**
338     * Sets the response for the most recent command
339     */
340     private void getResponse() {
341
342         // Attempts to get a response from the network
343         try {
344             response = (String) netIn.readUnshared();
345         } catch (IOException | ClassNotFoundException e) {

```

```

345         System.err.println(e.getMessage());
346     }
347
348     // Checks if the response was a partial
349     if (this.response.equals(id + "," + AFRS.PARTIAL_REQUEST)) {
350         System.out.println("Partial Request, Please finish request");
351         partial = true;
352     } else {
353         String[] split = response.split(",");
354
355         // Checks if the response was a connection
356         if (split[0].equals("connect")) {
357             id = Integer.parseInt(split[1]);
358
359         } else if (split.length > 1) {
360             if (split[1].equals(Server.DISCONNECT_REQUEST)) {
361                 disconnect();
362             }
363         }
364         partial = false;
365         System.out.println(response);
366     }
367
368     super.setChanged();
369     super.notifyObservers(response);
370 }
371
372 /**
373  * Disconnects the client from the network
374  */
375 void disconnect() {
376     try {
377         out.close();
378         netIn.close();
379         socket.close();
380     } catch (IOException e) {
381         System.err.println(e.getMessage());
382     }
383 }
384
385 }
386
387 package Controller.Requests;
388
389 import Model.Managers.AirportManager;
390
391 /**
392  * This class performs all actions for finding an airports info
393  *
394  * @author Amber Harding
395  * @author Mark Vittozzi
396  */
397 public class AirportInfoRequest implements Request {
398
399     private String airportCode;
400     private AirportManager airportManager;
401     private int id;
402
403     /**
404      * The AirportInfoRequest Constructor
405      *
406      * @param command a string array representing the information from the user
407      * @param airportManager the airport manager to be used
408      */
409     public AirportInfoRequest(String[] command, AirportManager airportManager) {
410         this.airportCode = command[2];
411         this.id = Integer.parseInt(command[0]);
412         this.airportManager = airportManager;
413     }

```

```

414     }
415
416     /**
417     * gets required airport information from AirportManager
418     *
419     * @return info: information to display about intended airport.
420     */
421     @Override
422     public String execute() {
423         return airportManager.getAirportInfo(airportCode, id);
424     }
425
426
427 }
428
429 package Controller.Requests;
430
431 import Model.Components.Itinerary;
432 import Model.Components.Passenger;
433 import Model.Managers.ReservationManager;
434
435 /**
436 * This class performs all actions for deleting a reservation
437 *
438 * @author Amber Harding
439 * @author Mark Vittozzi
440 * @author Ian Randman
441 */
442 public class DeleteReservationRequest implements UndoableRequest {
443     private final static String DELETE_SUCCESSFUL = "delete successful";
444     private final static String DELETE_ERROR = "error reservation not found";
445
446     private Passenger passenger;
447     private ReservationManager reservationManager;
448     private String origin;
449     private String destination;
450     private Itinerary itinerary;
451
452     /**
453     * The DeleteReservationRequest Constructor
454     *
455     * @param command          a string array representing the information from the
456     *                           user
457     * @param reservationManager the reservation manager to be used
458     */
459     public DeleteReservationRequest(String[] command, ReservationManager
460     reservationManager) {
461         this.passenger = new Passenger(command[2]);
462         this.reservationManager = reservationManager;
463         this.origin = command[3];
464         this.destination = command[4];
465     }
466
467     /**
468     * Has the reservation manager delete intended reservation
469     *
470     * @return a string indicating if the reservation deletion was successful
471     */
472     @Override
473     public String execute() {
474         itinerary = reservationManager.getReservations(passenger, origin,
475         destination).get(0).getItinerary();
476
477         if (reservationManager.removeReservation(passenger, origin, destination))
478             return DELETE_SUCCESSFUL;
479         else
480             return DELETE_ERROR;
481     }
482 }

```

```

480     /**
481     * Has the reservation manager undo the deletion of a reservation
482     *
483     * @return a message containing the passenger and itinerary
484     */
485     @Override
486     public String undo() {
487         reservationManager.addReservation(passenger, itinerary);
488         return "delete," + passenger.getName() + "," + itinerary;
489     }
490 }
491
492 package Controller.Requests;
493
494 import Controller.Sorting.AirfareSort;
495 import Controller.Sorting.ArrivalSort;
496 import Controller.Sorting.DepartureSort;
497 import Controller.Sorting.Sort;
498 import Model.Components.Airport;
499 import Model.Components.Itinerary;
500 import Model.Managers.AirportManager;
501 import Model.Managers.FlightManager;
502
503 import java.util.List;
504
505 /**
506  * This class performs all actions for obtaining flight information
507  *
508  * @author Amber Harding
509  * @author Matt Antantis
510  * @author Mark Vittozzi
511  * @author Ian Randman
512  * @author Jonathon Chierchio
513  */
514
515 public class FlightInfoRequest implements Request {
516
517     private static final String ORIGIN_ERROR = "error,unknown origin";
518     private static final String DESTINATION_ERROR = "error,unknown destination";
519     private static final String CONNECTION_ERROR = "error,invalid connection limit";
520     private static final String SORT_ERROR = "error,invalid sort order";
521
522     private static final String DEPARTURE_SORT = "departure";
523     private static final String ARRIVAL_SORT = "arrival";
524     private static final String AIRFARE_SORT = "airfare";
525
526     private String origin;
527     private String destination;
528     private int connections;
529     private String sortOrder;
530     private AirportManager airportManager;
531     private FlightManager flightManager;
532     private List<Itinerary> requestedItineraries;
533
534     /**
535     * The constructor for FlightInfoRequest
536     *
537     * @param command a string array representing the information from the user
538     * @param airportManager the airport manager needed for this task
539     * @param flightManager the flight manager needed for this task
540     */
541     public FlightInfoRequest(String[] command, AirportManager airportManager,
542                             FlightManager flightManager) {
543
544         this.airportManager = airportManager;
545         this.flightManager = flightManager;
546         this.origin = command[2];
547         this.destination = command[3];

```



```

548         if (command.length > 4 && !command[4].equals("")) {
549             try {
550                 this.connections = Integer.parseInt(command[4]);
551             } catch (ArrayIndexOutOfBoundsException e) {
552                 this.connections = 2;
553             }
554         } else {
555             this.connections = 2;
556         }
557
558         if (command.length == 6 && !command[5].equals("")) {
559             sortOrder = command[5];
560         } else {
561             sortOrder = DEPARTURE_SORT;
562         }
563     }
564
565     /**
566     * Receives and returns the flight info from the managers
567     *
568     * @return the pertinent flight information
569     */
570     public String execute() {
571         Airport originAirport = airportManager.getAirport(origin);
572         Airport destinationAirport = airportManager.getAirport(destination);
573         StringBuilder resp;
574
575         //Error checking
576         if (originAirport == null) {
577             //Checks that origin airport exists
578
579             resp = new StringBuilder(ORIGIN_ERROR);
580
581         } else if (destinationAirport == null) {
582             //Checks that destination airport exists
583
584             resp = new StringBuilder(DESTINATION_ERROR);
585
586         } else if (connections < 0
587             || connections > 2) {
588             //Checks connection limit is valid
589
590             resp = new StringBuilder(CONNECTION_ERROR);
591
592         } else if (!isValidSortOrder(sortOrder)) {
593             //Checks sort order is valid
594
595             resp = new StringBuilder(SORT_ERROR);
596
597         } else {
598             List<Itinerary> itineraries = flightManager.getPotentialItineraries(origin,
599                 destination, connections);
600
601             resp = new StringBuilder("info," + itineraries.size());
602             int count = 1;
603             requestedItineraries = itineraries;
604             sortItineraries(requestedItineraries);
605
606             for (Itinerary itinerary : requestedItineraries) {
607                 resp.append("\n").append(count).append(",").append(itinerary.getData());
608                 count++;
609             }
610
611             return resp.toString();
612
613         }
614
615     private boolean isValidSortOrder(String sortOrder) {

```

```

616         return sortOrder.equals(DEPARTURE_SORT) ||
617             sortOrder.equals(ARRIVAL_SORT) ||
618             sortOrder.equals(AIRFARE_SORT);
619     }
620
621     public List<Itinerary> getRequestedItineraries() {
622         return requestedItineraries;
623     }
624
625     /**
626      * Method used to sort itineraries based on the sortOrder
627      *
628      * @param itineraries list of itineraries being sorted
629      */
630     private void sortItineraries(List<Itinerary> itineraries) {
631         switch (sortOrder) {
632             case ARRIVAL_SORT:
633                 Sort Arrival = new ArrivalSort();
634                 Arrival.sort(itineraries);
635                 break;
636             case DEPARTURE_SORT:
637                 Sort Departure = new DepartureSort();
638                 Departure.sort(itineraries);
639                 break;
640             case AIRFARE_SORT:
641                 Sort Airfare = new AirfareSort();
642                 Airfare.sort(itineraries);
643                 break;
644         }
645     }
646 }
647
648 package Controller.Requests;
649
650 import Model.Components.Itinerary;
651 import Model.Components.Passenger;
652 import Model.Managers.ReservationManager;
653
654 import java.util.List;
655
656 /**
657  * This class performs all actions for making a reservation
658  *
659  * @author Amber Harding
660  * @author Matt Antantis
661  * @author Mark Vittozzi
662  * @author Ian Randman
663  */
664 public class MakeReservationRequest implements UndoableRequest {
665
666     private static final String RESERVE_SUCCESSFUL = "reserve,successful";
667     private static final String DUPLICATE_ERROR = "error, duplicate reservation";
668     private static final String INVALID_ID_ERROR = "error, invalid id";
669
670     private int id;
671     private Passenger passenger;
672     private ReservationManager reservationManager;
673     private List<Itinerary> requestedItineraries;
674     private String origin;
675     private String destination;
676     private Itinerary itinerary;
677
678     /**
679      * Constructor for MakeReservationRequest
680      *
681      * @param command          a string array representing the information from the
682                             user
683      * @param reservationManager The reservation manager needed for this request
684      * @param requestedItineraries The list of itineraries that a reservation will be

```

```

        selected from
684     */
685     public MakeReservationRequest(String[] command, ReservationManager
reservationManager,
686                                     List<Itinerary> requestedItineraries) {
687         this.reservationManager = reservationManager;
688         this.requestedItineraries = requestedItineraries;
689         this.id = Integer.parseInt(command[2]);
690         this.passenger = new Passenger(command[3]);
691     }
692
693
694
695     /**
696     * Has reservation manager make a reservation from the list of itineraries
697     *
698     * @return a string that indicates if the reservation was successfully created
699     */
700     @Override
701     public String execute() {
702         String info;
703         try {
704             itinerary = requestedItineraries.get(id - 1);
705             origin = itinerary.getOrigin();
706             destination = itinerary.getDestination();
707             boolean status = reservationManager.addReservation(passenger, itinerary);
708             if (status)
709                 info = RESERVE_SUCCESSFUL;
710             else
711                 info = DUPLICATE_ERROR;
712
713
714             } catch (IndexOutOfBoundsException e) {
715                 info = INVALID_ID_ERROR;
716             }
717             return info;
718         }
719
720     /**
721     * Has the reservation manager undo the making of a reservation
722     *
723     * @return a message containing the passenger and itinerary unreserved
724     */
725     @Override
726     public String undo() {
727         reservationManager.removeReservation(passenger, origin, destination);
728
729         return "make," + passenger.getName() + "," + itinerary;
730     }
731 }
732
733
734 package Controller.Requests;
735
736 /**
737  * @author Mark Vittozzi
738  * @author Amber Harding
739  */
740 public interface Request {
741     String execute();
742 }
743
744 package Controller.Requests;
745
746 import Model.Components.Passenger;
747 import Model.Components.Reservation;
748 import Model.Managers.ReservationManager;
749
750 import java.util.List;

```

```

751
752 /**
753  * This class is a command that will retrieve a reservation
754  *
755  * @author Amber Harding
756  * @author Mark Vittozzi
757  * @author Ian Randman
758  */
759
760 public class RetrieveReservationRequest implements Request {
761
762     private Passenger passenger;
763     private ReservationManager reservationManager;
764     private String origin;
765     private String destination;
766
767     /**
768      * The RetrieveReservationRequest constructor
769      *
770      * @param command the command to be processed
771      */
772     public RetrieveReservationRequest(String[] command, ReservationManager
reservationManager) {
773         this.passenger = new Passenger(command[2]);
774         this.reservationManager = reservationManager;
775         if (command.length == 4) {
776             try {
777                 this.origin = "";
778                 this.destination = command[3];
779             } catch (ArrayIndexOutOfBoundsException e) {
780                 System.out.println(e.getMessage());
781             }
782         } else if (command.length == 5) {
783             this.origin = command[3];
784             this.destination = command[4];
785         } else {
786             this.origin = "";
787             this.destination = "";
788         }
789     }
790
791     @Override
792     public String execute() {
793
794         List<Reservation> reservations = reservationManager.getReservations(passenger,
origin, destination);
795
796         StringBuilder info = new StringBuilder("retrieve," + reservations.size());
797
798         for (Reservation r : reservations) {
799             info.append(r.getData());
800         }
801
802         return info.toString();
803     }
804
805 }
806
807
808
809 package Controller.Requests;
810
811 import Model.Managers.AirportManager;
812
813 /**
814  * This class performs all actions for changing the info server a client is using
815  *
816  * @author Mark Vittozzi
817  */

```

```

818
819 public class SetInfoModuleRequest implements Request {
820
821
822     private String server;
823     private AirportManager airportManager;
824     private int id;
825
826     /**
827      * The AirportInfoRequest Constructor
828      *
829      * @param command      a string array representing the information from the user
830      * @param airportManager the airport manager to be used
831      */
832     public SetInfoModuleRequest(String[] command, AirportManager airportManager) {
833         this.server = command[2];
834         this.id = Integer.parseInt(command[0]);
835         this.airportManager = airportManager;
836     }
837
838     /**
839      * Calls the switch client method on its airport manager and returns a string
840      * alerting the client that the server was successfully changed
841      */
842     @Override
843     public String execute() {
844         airportManager.switchClientModule(id, server);
845         return "Server,Successful";
846     }
847 }
848
849
850 package Controller.Requests;
851
852 /**
853  * Interface for requests that can be undone
854  *
855  * @author Ian Randman
856  */
857 public interface UndoableRequest extends Request {
858     String undo();
859 }
860
861 package Controller.Requests;
862
863 import Controller.Server.Server;
864
865 import java.util.Stack;
866
867 /**
868  * @author Ian Randman
869  * @author Matt Antantis
870  */
871 public class UndoRedoRequestHandler {
872
873     private final static String ERROR_MESSAGE = "error,no request available";
874
875     private Stack<UndoableRequest> undoStack;
876     private Stack<UndoableRequest> redoStack;
877
878     /**
879      * The constructor for the UndoRedoRequestHandler
880      */
881     public UndoRedoRequestHandler() {
882         this.undoStack = new Stack<>();
883         this.redoStack = new Stack<>();
884     }
885
886     /**

```

```

887     * Adds a request to the stack of undoable requests
888     *
889     * @param request the command to add to the undo stack
890     */
891     public void addRequest(UndoableRequest request) {
892         undoStack.push(request);
893     }
894
895     /**
896     * Undoes the most recent command in the undo stack
897     * and adds it to the redo stack
898     *
899     * @return a message containing the status of the undo action
900     */
901     public String undo() {
902         if (!undoStack.empty()) {
903             UndoableRequest request = undoStack.pop();
904             redoStack.push(request);
905             return Server.UNDO_REQUEST + "," + request.undo();
906         }
907
908         return ERROR_MESSAGE;
909     }
910
911     /**
912     * Performs the most recent command in the redo stack
913     * and adds it to the undo stack
914     *
915     * @return a message containing the status of the redo action
916     */
917     public String redo() {
918         if (!redoStack.empty()) {
919             UndoableRequest request = redoStack.pop();
920             undoStack.push(request);
921             return Server.REDO_REQUEST + "," + request.execute();
922         }
923
924         return ERROR_MESSAGE;
925     }
926 }
927
928 package Controller.Server;
929
930 import Controller.Requests.*;
931 import Model.Components.Itinerary;
932 import Model.Managers.AirportManager;
933 import Model.Managers.FlightManager;
934 import Model.Managers.ReservationManager;
935
936 import java.util.HashMap;
937 import java.util.List;
938 import java.util.Map;
939
940 /**
941  * Coordinates requests from the client
942  * and finds the necessary information
943  *
944  * @author Amber Harding
945  * @author Mark Vittozzi
946  * @author Ian Randman
947  * @author Matt Antantis
948  */
949 public class AFRS implements Server {
950
951     private AirportManager airportManager;
952     private FlightManager flightManager;
953     private ReservationManager reservationManager;
954
955

```

```

956 private Map<Integer, List<Itinerary>> itineraryMap;
957 private Map<Integer, UndoRedoRequestHandler> undoRedoRequestHandlerMap;
958
959 private int clientIDIndex;
960
961
962 /**
963  * Construction for Controller.Server.AFRS. Initializes Model.Managers
964  */
965 public AFRS() {
966     this.airportManager = new AirportManager();
967     this.flightManager = new FlightManager(airportManager);
968     this.reservationManager = new ReservationManager();
969
970     this.itineraryMap = new HashMap<>();
971     this.undoRedoRequestHandlerMap = new HashMap<>();
972     this.clientIDIndex = 1;
973
974 }
975
976 /**
977  * Adds a connection to the map of connections
978  */
979 synchronized int addClient() {
980     itineraryMap.put(clientIDIndex, null);
981     undoRedoRequestHandlerMap.put(clientIDIndex, new UndoRedoRequestHandler());
982     clientIDIndex++;
983     return clientIDIndex - 1;
984 }
985
986 /**
987  * Reads in a string, from this a request is created and executed.
988  *
989  * @param command a string array that is used to create and execute a request
990  */
991 public synchronized String runCommand(String[] command) {
992     Request request;
993     String response = "";
994     int index = Integer.parseInt(command[0]);
995
996     switch (command[1]) {
997         case INFO_REQUEST:
998             //create flight info request
999             request = new FlightInfoRequest(command, airportManager, flightManager);
1000             response = request.execute();
1001             itineraryMap.put(index, ((FlightInfoRequest)
1002                 request).getRequestedItineraries());
1003             request = null;
1004             break;
1005
1006         case RESERVE_REQUEST:
1007             //create reservation request
1008             request = new MakeReservationRequest(command, reservationManager,
1009                 itineraryMap.get(index));
1009
1010             undoRedoRequestHandlerMap.get(index).addRequest((MakeReservationRequest)
1011                 request);
1012             break;
1013
1014         case RETRIEVE_REQUEST:
1015             //create retrieve reservation request
1016             request = new RetrieveReservationRequest(command, reservationManager);
1017             break;
1018
1019         case DELETE_REQUEST:
1020             //create delete reservation request
1021             request = new DeleteReservationRequest(command, reservationManager);

```

```

        undoRedoRequestHandlerMap.get(index).addRequest((DeleteReservationRequest
        ) request);
1021     break;
1022
1023     case AIRPORT_REQUEST:
1024         //create airport request
1025         request = new AirportInfoRequest(command, airportManager);
1026         break;
1027
1028     case DISCONNECT_REQUEST:
1029         itineraryMap.remove(index);
1030         undoRedoRequestHandlerMap.remove(index);
1031         System.out.println(index + " disconnected");
1032         return DISCONNECT_REQUEST;
1033
1034     case UNDO_REQUEST:
1035         request = null;
1036         response = undoRedoRequestHandlerMap.get(index).undo();
1037         break;
1038
1039     case REDO_REQUEST:
1040         request = null;
1041         response = undoRedoRequestHandlerMap.get(index).redo();
1042         break;
1043     case CHANGE_MODULE_REQUEST:
1044         request = new SetInfoModuleRequest(command, airportManager);
1045         break;
1046
1047     default:
1048         request = null;
1049         response = "error, Unknown request";
1050         break;
1051     }
1052
1053     if (request != null) {
1054         response = request.execute();
1055     }
1056
1057     return response;
1058 }
1059 }
1060
1061 package Controller.Server;
1062
1063 import java.io.IOException;
1064 import java.net.ServerSocket;
1065
1066 /**
1067  * The Proxy for all commands being sent to the server
1068  *
1069  * @author Ian Randman
1070  * @author Matt Antantis
1071  */
1072 public class ProxyServer implements Server {
1073
1074     private final static int PORT = 4567;
1075     private AFRS afrs;
1076     private ServerSocket serverSocket;
1077
1078     /**
1079      * The constructor for the ProxyServer
1080      *
1081      * @param afrs the real server to defer commands to
1082      */
1083     public ProxyServer(AFRS afrs) {
1084         this.afrs = afrs;
1085         try {
1086             serverSocket = new ServerSocket(PORT);
1087         } catch (IOException e) {

```



```

1088         System.err.println(e.getMessage());
1089     }
1090 }
1091
1092 /**
1093  * Runs the proxy server
1094  */
1095 public void run() {
1096
1097     // While the client is connected
1098     while (!serverSocket.isClosed()) {
1099         try {
1100             ServerThread temp = new ServerThread(serverSocket.accept(), this);
1101             temp.start();
1102             System.out.println("new connection made");
1103         } catch (IOException e) {
1104             System.err.println(e.getMessage());
1105         }
1106     }
1107 }
1108
1109
1110 /**
1111  * Executes a command from the client
1112  *
1113  * @param command the command to be executed
1114  * @return the response from the command
1115  */
1116 @Override
1117 public synchronized String runCommand(String[] command) {
1118
1119     String response;
1120     int index;
1121
1122     String end = command[command.length - 1];
1123
1124     if (!command[0].contains(CONNECT_REQUEST) ||
1125         command[0].contains(DISCONNECT_REQUEST)) {
1126         index = Integer.parseInt(command[0]);
1127     } else {
1128         index = 0;
1129     }
1130
1131     // Test if command is not complete
1132     if (end.charAt(end.length() - 1) != ';') {
1133         return index + "," + PARTIAL_REQUEST;
1134     } else {
1135         // Cleans the command for the server
1136         command = getCorrectCommand(command);
1137
1138         // Checks if client is not connected and wants to connect
1139         if (command[0].equals(CONNECT_REQUEST)) {
1140             index = afrs.addClient();
1141             response = Server.CONNECT_REQUEST + "," + index;
1142         }
1143         // Checks if the user is not connected
1144         else if (index == 0) {
1145             response = "error,invalid connection";
1146         }
1147         // Checks if the user is connected and wants to connect again
1148         else if (command[1].equals(CONNECT_REQUEST)) {
1149             response = "error,connection limit reached";
1150         }
1151         // Otherwise, the command can be sent to the server
1152         else {
1153             response = index + "," + afrs.runCommand(command);
1154         }
1155     }
1156     return response;

```

```

1156     }
1157
1158     /**
1159     * Returns command string array without the semicolon at the end
1160     *
1161     * @param command a string array containing input
1162     * @return the same string array with the last char from the last index removed
1163     */
1164     private String[] getCorrectCommand(String[] command) {
1165         String holder = command[command.length - 1];
1166         command[command.length - 1] = holder.substring(0, holder.length() - 1);
1167         return command;
1168     }
1169 }
1170
1171 package Controller.Server;
1172
1173 /**
1174 * The interface for the proxy and subject servers
1175 *
1176 * @author Ian Randman
1177 * @author Matt Antantis
1178 * @author Mark Vittozzi
1179 */
1180 public interface Server {
1181
1182     String PARTIAL_REQUEST = "partial-request";
1183     String INFO_REQUEST = "info";
1184     String RESERVE_REQUEST = "reserve";
1185     String RETRIEVE_REQUEST = "retrieve";
1186     String DELETE_REQUEST = "delete";
1187     String AIRPORT_REQUEST = "airport";
1188     String CONNECT_REQUEST = "connect";
1189     String DISCONNECT_REQUEST = "disconnect";
1190     String UNDO_REQUEST = "undo";
1191     String REDO_REQUEST = "redo";
1192     String CHANGE_MODULE_REQUEST = "server";
1193
1194     String runCommand(String[] command);
1195 }
1196
1197 package Controller.Server;
1198
1199 import java.io.IOException;
1200 import java.io.ObjectInputStream;
1201 import java.io.ObjectOutputStream;
1202 import java.net.Socket;
1203
1204 /**
1205 * This class allows the server to always be listening for requests coming in from
1206 * clients. To accomplish this, it is
1207 * run as a thread.
1208 */
1209 public class ServerThread extends Thread {
1210     private Server server;
1211     private Socket socket;
1212     private ObjectInputStream netIn;
1213     private ObjectOutputStream out;
1214
1215     /**
1216     * Set up the input and output streams for the client-server connection.
1217     *
1218     * @param socket the server's connection to the client
1219     * @param server the main server
1220     */
1221     ServerThread(Socket socket, Server server) {
1222         this.server = server;
1223         this.socket = socket;
1224     }

```

```

1224         try {
1225             this.netIn = new ObjectInputStream(socket.getInputStream());
1226             this.out = new ObjectOutputStream(socket.getOutputStream());
1227         } catch (IOException e) {
1228             System.err.println(e.getMessage());
1229         }
1230     }
1231
1232     /**
1233     * Alternate between receiving a request from the Client and send a response back
1234     * to the Client.
1235     */
1236     @Override
1237     public void run() {
1238         try {
1239             while (!socket.isClosed()) {
1240                 String[] command = (String[]) (netIn.readUnshared());
1241
1242                 String response = server.runCommand(command);
1243
1244                 String[] brokenResponse = response.split(",");
1245                 out.writeUnshared(response);
1246
1247                 if (brokenResponse[1].equals(Server.DISCONNECT_REQUEST)) {
1248                     out.close();
1249                     netIn.close();
1250                     socket.close();
1251                 }
1252             }
1253         } catch (IOException | ClassNotFoundException e) {
1254             System.err.println(e.getMessage());
1255         }
1256     }
1257 }
1258
1259 package Controller.Sorting;
1260
1261 import Model.Components.Itinerary;
1262
1263 import java.util.List;
1264
1265 /**
1266  * Algorithm for sorting the flight itineraries based on airfare
1267  *
1268  * @author Jonathon Chierchio
1269  * @author Ian Randman
1270  */
1271 public class AirfareSort implements Sort {
1272
1273     /**
1274     * method for performing a sort on total airfare of itineraries
1275     *
1276     * @param itineraries list of itineraries being sorted
1277     */
1278     public void sort(List<Itinerary> itineraries) {
1279
1280         int length = itineraries.size();
1281
1282         for (int index = 0; index < length; index++) {
1283             int pos = index;
1284             for (int j = index; j < length; j++) {
1285                 if (itineraries.get(j).getAirfare() < itineraries.get(pos).getAirfare())
1286                     pos = j;
1287             }
1288
1289             Itinerary min = itineraries.get(pos);
1290             itineraries.set(pos, itineraries.get(index));
1291             itineraries.set(index, min);

```

```

1292     }
1293 }
1294
1295 }
1296 }
1297
1298 package Controller.Sorting;
1299
1300 import Model.Components.Itinerary;
1301
1302 import java.util.List;
1303
1304 /**
1305  * Algorithm for sorting the flight itineraries based on arrival time
1306  *
1307  * @author Jonathon Chierchio
1308  * @author Ian Randman
1309  */
1310 public class ArrivalSort implements Sort {
1311
1312     /**
1313      * method for performing a sort on arrival times
1314      *
1315      * @param itineraries list of itineraries being sorted
1316      */
1317     public void sort(List<Itinerary> itineraries) {
1318
1319         int length = itineraries.size();
1320
1321         for (int index = 0; index < length; index++) {
1322             int pos = index;
1323             for (int j = index; j < length; j++) {
1324                 if (itineraries.get(j).getArrivalTime() <
1325                     itineraries.get(pos).getArrivalTime())
1326                     pos = j;
1327             }
1328
1329             Itinerary min = itineraries.get(pos);
1330             itineraries.set(pos, itineraries.get(index));
1331             itineraries.set(index, min);
1332         }
1333     }
1334 }
1335
1336 package Controller.Sorting;
1337
1338 import Model.Components.Itinerary;
1339
1340 import java.util.List;
1341
1342 /**
1343  * Algorithm for sorting the flight itineraries based on departure time
1344  *
1345  * @author Jonathon Chierchio
1346  * @author Ian Randman
1347  */
1348 public class DepartureSort implements Sort {
1349
1350     /**
1351      * method for performing a sort on departure times
1352      *
1353      * @param itineraries list of itineraries being sorted
1354      */
1355     public void sort(List<Itinerary> itineraries) {
1356
1357         int length = itineraries.size();
1358
1359         for (int index = 0; index < length; index++) {

```

```

1360         int pos = index;
1361         for (int j = index; j < length; j++) {
1362             if (itineraries.get(j).getDepartureTime() <
                itineraries.get(pos).getDepartureTime())
                pos = j;
1363         }
1364     }
1365
1366     Itinerary min = itineraries.get(pos);
1367     itineraries.set(pos, itineraries.get(index));
1368     itineraries.set(index, min);
1369
1370 }
1371 }
1372 }
1373
1374 package Controller.Sorting;
1375
1376 import Model.Components.Itinerary;
1377
1378 import java.util.List;
1379
1380 /**
1381  * Deals with the sorting algorithm needed to sort flight itineraries
1382  *
1383  * @author Amber Harding
1384  * @author Ian Randman
1385  */
1386 public interface Sort {
1387
1388     void sort(List<Itinerary> itineraries);
1389 }
1390
1391 package Model.Components.Weather;
1392
1393 /**
1394  * A object to hold information from JSON
1395  *
1396  * @author Mark Vittozzi
1397  */
1398 public class Status implements Comparable {
1399     private String Type;
1400     private String Reason;
1401     private String AvgDelay;
1402     private String EndTime;
1403     private String MinDelay;
1404     private String MaxDelay;
1405
1406     /**
1407      * Determines which passed in strings are null
1408      * Non null strings are added to return statement
1409      *
1410      * @return a string representing this delay
1411      */
1412     @Override
1413     public String toString() {
1414
1415         String type = "";
1416         String reason = "";
1417         String avg = "";
1418         String end = "";
1419         String minDelay = "";
1420         String maxDelay = "";
1421
1422         if (Type != null)
1423             type = "Type: " + Type + ", ";
1424         if (Reason != null && !Reason.equals(""))
1425             reason = "Reason " + reason + ", ";
1426         if (AvgDelay != null)
1427             avg = "AvgDelay: " + AvgDelay + ", ";

```

```

1428         if (EndTime != null)
1429             end = "EndTime: " + EndTime + ", ";
1430         if (MinDelay != null)
1431             minDelay = "MinTime: " + MinDelay + ", ";
1432         if (MaxDelay != null)
1433             maxDelay = "MaxTime: " + MaxDelay;
1434         return type + reason + avg + end + minDelay + maxDelay;
1435     }
1436 }
1437
1438
1439 @Override
1440 public int compareTo(Object o) {
1441     if (o instanceof Status) {
1442         return Integer.parseInt(this.MaxDelay) - Integer.parseInt(((Status)
1443             o).MaxDelay);
1444     } else
1445         return 0;
1446 }
1447
1448 package Model.Components.Weather;
1449
1450 import java.util.ArrayList;
1451 import java.util.List;
1452
1453 /**
1454  * A class for storing and returning weather for an airport
1455  *
1456  * @author Mark Vittozzi
1457  */
1458
1459 public class StoredWeather implements WeatherModule {
1460
1461     private List<String> storedWeather;
1462     private String delay;
1463
1464     public StoredWeather() {
1465         storedWeather = new ArrayList<>();
1466         this.delay = "";
1467     }
1468
1469     /**
1470      * @param delay: a string representing the delay at this airport
1471      */
1472     public void setDelay(String delay) {
1473         this.delay = delay;
1474     }
1475
1476     /**
1477      * Adds a string representing weather to the list containing
1478      * all weathers
1479      *
1480      * @param weather: A string representing the weather
1481      */
1482     @Override
1483     public void addWeather(String weather) {
1484         storedWeather.add(weather);
1485     }
1486
1487     /**
1488      * Checks if the index variable is equal to the size of the weather list
1489      * this is used for making sure the index variable does not go over
1490      * the length-1
1491      *
1492      * @param index: the index variable used by the object who holds this class
1493      * @return a boolean representing if the index is equal to the list length
1494      */
1495     @Override

```

```

1496     public boolean checkId(int index) {
1497         return (index == storedWeather.size());
1498     }
1499
1500     /**
1501     * returns the weather string at the specified index
1502     *
1503     * @param index the index of the string to be returned
1504     * @return the weather string
1505     */
1506     @Override
1507     public String getWeather(int index) {
1508         return storedWeather.get(index) + "," + delay;
1509     }
1510 }
1511
1512 package Model.Components.Weather;
1513
1514 import java.util.List;
1515 /**
1516  * An object to hold information from JSON
1517  *
1518  * @author Mark Vittozzi
1519  */
1520
1521 public class Weather {
1522
1523     List<String> Temp;
1524
1525     /**
1526     * @return a string representing the temp list
1527     */
1528     public String getTemp() {
1529         String resp = " Temperature: ";
1530         for(String t : Temp){
1531             resp += t + " ";
1532         }
1533         return resp;
1534     }
1535
1536     /**
1537     * @return the string representing the temp list
1538     */
1539     @Override
1540     public String toString() {
1541         return getTemp();
1542     }
1543 }
1544
1545
1546 }
1547
1548 package Model.Components.Weather;
1549
1550 /**
1551  * A interface for the two weather module classes to implement
1552  * (Stored weather and web weather)
1553  *
1554  * @author Mark Vittozzi
1555  */
1556
1557 public interface WeatherModule {
1558
1559     String getWeather(int id);
1560
1561     void addWeather(String weather);
1562
1563     boolean checkId(int id);
1564

```

```

1565         void setDelay(String delay);
1566     }
1567
1568     package Model.Components.Weather;
1569
1570     import java.util.Collections;
1571     import java.util.List;
1572
1573     /**
1574      * An object to hold information from JSON
1575      *
1576      * @author Mark Vittozzi
1577      */
1578
1579
1580     public class WebAirport {
1581         private String Name;
1582         private int DelayCount;
1583         private Weather Weather;
1584         private List<Status> Status;
1585
1586         /**
1587          * @return the list of statuses as a single string
1588          */
1589         private String getStatus() {
1590             StringBuilder statusString = new StringBuilder();
1591             for (int i = 0; i < Status.size(); i++) {
1592                 if (i == Status.size() - 1)
1593                     statusString.append(i + 1).append(" ").append(Status.get(i));
1594                 else
1595                     statusString.append(i + 1).append("
").append(Status.get(i)).append("\n");
1596             }
1597             return statusString.toString();
1598         }
1599
1600
1601         /**
1602          * @return a string representing the class
1603          */
1604         @Override
1605         public String toString() {
1606             Collections.sort(Status);
1607             if (DelayCount > 0)
1608                 return Name + "," + Weather + ",Delay count: " + DelayCount + "\nDelays:
" + getStatus();
1609             else
1610                 return Name + "," + Weather + ",Delay count: " + DelayCount;
1611         }
1612     }
1613
1614     package Model.Components.Weather;
1615
1616     import com.google.gson.Gson;
1617
1618     import java.io.BufferedReader;
1619     import java.io.IOException;
1620     import java.io.InputStreamReader;
1621     import java.net.HttpURLConnection;
1622     import java.net.MalformedURLException;
1623     import java.net.ProtocolException;
1624     import java.net.URL;
1625
1626     /**
1627      * A class to gather weather information from the web for an airport
1628      *
1629      * @author Mark Vittozzi
1630      */
1631

```



```

1632
1633 public class WebWeather implements WeatherModule {
1634
1635     // the URL of the site the weather info is stored
1636     private static final String URL_PREFACE =
1637         "https://soa.smext.faa.gov/asws/api/airport/status/";
1638
1639     private String airportCode;
1640
1641     /**
1642      * @param airportCode: Takes in an airport code, this is the airport the weather
1643      * info will be for
1644      */
1645     public WebWeather(String airportCode) {
1646         this.airportCode = airportCode;
1647     }
1648
1649     /**
1650      * @param id an id that is not used but required for interface
1651      * @return a string representing the weather for an airport
1652      */
1653     @Override
1654     public String getWeather(int id) {
1655         return generateWeather();
1656     }
1657
1658     // A method that is not used but required for interface
1659     @Override
1660     public void addWeather(String weather) {
1661     }
1662
1663     // A method that is not used but required for interface
1664     @Override
1665     public boolean checkId(int id) {
1666         return false;
1667     }
1668
1669     // A method that is not used but is required for interface
1670     @Override
1671     public void setDelay(String delay) {
1672     }
1673
1674     /**
1675      * Establishes a connection with the URL and obtains the weather info
1676      * stores them in objects using GSON
1677      *
1678      * @return a string representing the weather
1679      */
1680     private String generateWeather() {
1681
1682         String airline = URL_PREFACE + airportCode;
1683         WebAirport webAirport = null;
1684
1685         try {
1686             URL FAA_URL = new URL(airline);
1687             HttpURLConnection urlConnection = (HttpURLConnection)
1688                 FAA_URL.openConnection();
1689             urlConnection.setRequestMethod("GET");
1690             urlConnection.setConnectTimeout(10000);
1691             urlConnection.setReadTimeout(10000);
1692             urlConnection.setRequestProperty("Accept", "application/" + "json");
1693             BufferedReader in =
1694                 new BufferedReader(new
1695                     InputStreamReader(urlConnection.getInputStream()));
1696             String inputLine;
1697             StringBuilder response = new StringBuilder();
1698
1699             while ((inputLine = in.readLine()) != null) {

```

```

1697         response.append(inputLine);
1698     }
1699     Gson gson = new Gson();
1700
1701     webAirport = gson.fromJson(response.toString(), WebAirport.class);
1702     in.close();
1703
1704     } catch (MalformedURLException e) {
1705         System.out.print("Malformed URL: ");
1706         System.out.println(e.getMessage());
1707     } catch (ProtocolException e) {
1708         System.out.print("Unsupported protocol: ");
1709         System.out.println(e.getMessage());
1710     } catch (IOException e) {
1711         System.out.println(e.getMessage());
1712     }
1713
1714     return webAirport.toString();
1715 }
1716 }
1717
1718 package Model.Components;
1719
1720 import Model.Managers.WeatherManager;
1721
1722 /**
1723  * Stores the information for an airport
1724  *
1725  * @author Matt Antantis
1726  * @author Mark Vittozzi
1727  */
1728 public class Airport {
1729
1730     private String code;
1731     private String city;
1732     private int connectionTime;
1733     private int delay;
1734     private WeatherManager weatherManager;
1735
1736
1737     /**
1738      * The constructor for the airport object
1739      *
1740      * @param code the airport's three letter code
1741      * @param city the airport's city
1742      */
1743     public Airport(String code, String city) {
1744         this.code = code;
1745         this.city = city;
1746         this.weatherManager = new WeatherManager(code);
1747     }
1748
1749     /**
1750      * Sets the connection time for the airport
1751      *
1752      * @param connectionTime the time for connections
1753      */
1754     public void setConnectionTime(int connectionTime) {
1755         this.connectionTime = connectionTime;
1756     }
1757
1758     /**
1759      * Adds a weather report to the airport
1760      *
1761      * @param weather string containing the weather being added
1762      */
1763     public void addWeather(String weather) {
1764         weatherManager.addWeather(weather);
1765     }

```

```

1766
1767 /**
1768  * Gets the most up to date weather report for the airport
1769  *
1770  * @return up to date weather
1771  */
1772 public String getWeather(int id) {
1773     return weatherManager.getWeather(id);
1774 }
1775
1776
1777 /**
1778  * Sets the delay time for the airport
1779  *
1780  * @param delay the time of the delay
1781  */
1782 public void setDelay(int delay) {
1783     this.delay = delay;
1784     this.weatherManager.setDelay(String.valueOf(delay));
1785 }
1786
1787 /**
1788  * determines overlay at airports
1789  *
1790  * @return overlay
1791  */
1792 public int getOverlay() {
1793     return delay + connectionTime;
1794 }
1795
1796 /**
1797  * Determines if an object is equal to this
1798  *
1799  * @param o the object being compared
1800  * @return if the two objects are equal
1801  */
1802 public boolean equals(Object o) {
1803     if (o.getClass().equals(this.getClass())) {
1804         Airport a = (Airport) o;
1805         return a.code.equals(this.code);
1806     }
1807     return false;
1808 }
1809
1810 /**
1811  * Changes the weatherModule for the user
1812  *
1813  * @param clientId an int representing the users ID
1814  * @param server the server that the client wants to switch to
1815  */
1816 public void changeModule(int clientId, String server) {
1817     weatherManager.switchModules(clientId, server);
1818 }
1819 }
1820
1821 package Model.Components;
1822
1823 /**
1824  * Stores the data for a specific flight
1825  *
1826  * @author Matt Antantis
1827  * @author Ian Randman
1828  */
1829 public class Flight implements Trip {
1830
1831     private String originCode;
1832     private String destinationCode;
1833     private String departureTime;
1834     private String arrivalTime;

```

```

1835 private int flightNumber;
1836 private int airfare;
1837 private int numericArrival;
1838 private int numericDeparture;
1839
1840 /**
1841  * The constructor for Flight from an array of Strings
1842  *
1843  * @param flight the details for the flight
1844  */
1845 public Flight(String[] flight) {
1846     this.originCode = flight[0];
1847     this.destinationCode = flight[1];
1848     this.departureTime = flight[2];
1849     this.arrivalTime = flight[3];
1850     this.flightNumber = Integer.parseInt(flight[4]);
1851     this.airfare = Integer.parseInt(flight[5]);
1852     setNumericTimes();
1853 }
1854
1855 /**
1856  * Tests if the flight is between a requested origin and destination
1857  *
1858  * @param origin the origin airport code
1859  * @param destination the destination airport code
1860  * @return if the flight is between the two given airports
1861  */
1862 public boolean validFlight(String origin, String destination) {
1863     return this.originCode.equals(origin) &&
1864         this.destinationCode.equals(destination);
1865 }
1866
1867 /**
1868  * Gets the airfare cost for the flight
1869  *
1870  * @return the airfare for the flight
1871  */
1872 @Override
1873 public int getAirfare() {
1874     return airfare;
1875 }
1876
1877 /**
1878  * Gets the origin of the trip
1879  *
1880  * @return the code for the origin airport
1881  */
1882 @Override
1883 public String getOrigin() {
1884     return originCode;
1885 }
1886
1887 /**
1888  * Gets the destination of the trip
1889  *
1890  * @return the code for the destination airport
1891  */
1892 @Override
1893 public String getDestination() {
1894     return destinationCode;
1895 }
1896
1897 /**
1898  * converts string time (10:30a) into military time for easier use of sorting
1899  */
1900 private void setNumericTimes() {
1901     if (arrivalTime.charAt(arrivalTime.length() - 1) == 'a')
1902         numericArrival = 0;
1903     else

```

```

1903         numericArrival = 1200;
1904
1905         numericArrival += buildTime(arrivalTime);
1906
1907         if (departureTime.charAt(departureTime.length() - 1) == 'a')
1908             numericDeparture = 0;
1909         else
1910             numericDeparture = 1200;
1911
1912         numericDeparture += buildTime(departureTime);
1913     }
1914
1915     /**
1916     * helper function for setNumericTimes()
1917     *
1918     * @param time time of arrival or departures
1919     * @return int version of time
1920     */
1921     private int buildTime(String time) {
1922         int num = 0;
1923         for (int i = 0; i < time.length(); i++) {
1924             char value = time.charAt(i);
1925             if (value >= '0' && value <= '9') {
1926                 num += value - 48;
1927                 num *= 10;
1928             }
1929         }
1930         return (num / 10) % 1200;
1931     }
1932
1933     /**
1934     * returns numeric arrival time
1935     */
1936     public int getNumericArrival() {
1937         return numericArrival;
1938     }
1939
1940     /**
1941     * returns numeric departure time
1942     */
1943     public int getNumericDeparture() {
1944         return numericDeparture;
1945     }
1946
1947     /**
1948     * Gets the information to display to the user
1949     *
1950     * @return the flight data to show to the user
1951     */
1952     public String getData() {
1953         return flightNumber + "," + originCode + "," + departureTime + "," +
            destinationCode + "," + arrivalTime;
1954     }
1955
1956     /**
1957     * Converts the flight data into a string
1958     *
1959     * @return the string version of the flight
1960     */
1961     @Override
1962     public String toString() {
1963         return originCode + "," + destinationCode + "," + departureTime + "," +
            arrivalTime + "," + flightNumber + "," + airfare;
1964     }
1965 }
1966
1967 package Model.Components;
1968
1969 /**

```

```

1970  * This class stores the information for a series of flights
1971  * between two airports
1972  *
1973  * @author Matt Antantis
1974  * @author Ian Randman
1975  */
1976 public class Itinerary implements Trip {
1977
1978     private Flight[] flights;
1979
1980     /**
1981     * The constructor for the Itinerary
1982     *
1983     * @param flights the list of flights in the itinerary
1984     */
1985     public Itinerary(Flight[] flights) {
1986         this.flights = flights;
1987     }
1988
1989     /**
1990     * Gets the number of flights in the itinerary
1991     *
1992     * @return the number of flights
1993     */
1994     private int getNumFlights() {
1995         return flights.length;
1996     }
1997
1998     /**
1999     * Gets the total airfare cost for the trip
2000     *
2001     * @return the total airfare for the trip
2002     */
2003     @Override
2004     public int getAirfare() {
2005         int sum = 0;
2006         for (Flight f : flights) {
2007             sum += f.getAirfare();
2008         }
2009         return sum;
2010     }
2011
2012     /**
2013     * Gets the data to display to the user
2014     *
2015     * @return the data for the itinerary
2016     */
2017     public String getData() {
2018         StringBuilder result = new StringBuilder(getAirfare() + "," + getNumFlights());
2019         for (Flight f : flights) {
2020             result.append(",").append(f.getData());
2021         }
2022         return result.toString();
2023     }
2024
2025     /**
2026     * Gets the origin of the trip
2027     *
2028     * @return the code for the starting airport
2029     */
2030     @Override
2031     public String getOrigin() {
2032         return flights[0].getOrigin();
2033     }
2034
2035     /**
2036     * Gets the destination of the trip
2037     *
2038     * @return the code for the final airport

```

```

2039     */
2040     @Override
2041     public String getDestination() {
2042         return flights[flights.length - 1].getDestination();
2043     }
2044
2045     /**
2046     * Converts the itinerary data into a string
2047     *
2048     * @return the string version of the itinerary
2049     */
2050     @Override
2051     public String toString() {
2052         StringBuilder result = new StringBuilder("" + getNumFlights());
2053         for (Flight f : flights) {
2054             result.append(",").append(f.toString());
2055         }
2056         return result.toString();
2057     }
2058
2059     /**
2060     * obtains the arrival time for the final flight in the itinerary
2061     *
2062     * @return integer time of arrival for the itinerary
2063     */
2064     public int getArrivalTime() {
2065         return flights[flights.length - 1].getNumericArrival();
2066     }
2067
2068     /**
2069     * obtains the departure time of an itinerary
2070     *
2071     * @return integer time of departure for the itinerary
2072     */
2073     public int getDepartureTime() {
2074         return flights[0].getNumericDeparture();
2075     }
2076 }
2077
2078 package Model.Components;
2079
2080 /**
2081  * This class stores data for the Passenger
2082  * for a reservation
2083  *
2084  * @author Matt Antantis
2085  */
2086 public class Passenger {
2087     private String name;
2088
2089     /**
2090     * The constructor for Passenger
2091     *
2092     * @param name the name of the Passenger
2093     */
2094     public Passenger(String name) {
2095         this.name = name;
2096     }
2097
2098     /**
2099     * Gets the name of the passenger
2100     *
2101     * @return the passenger's name
2102     */
2103     public String getName() {
2104         return name;
2105     }
2106
2107     /**

```

```

2108     * Determines if this passenger is equal to another object
2109     *
2110     * @param o the object being compared
2111     * @return if the two objects are equal
2112     */
2113     @Override
2114     public boolean equals(Object o) {
2115         if (o == null || getClass() != o.getClass())
2116             return false;
2117
2118         Passenger that = (Passenger) o;
2119
2120         return this.name.equals(that.name);
2121     }
2122 }
2123
2124 package Model.Components;
2125
2126 /**
2127  * This class stores the reservation for a passenger and an itinerary
2128  *
2129  * @author Matt Antantis
2130  * @author Ian Randman
2131  */
2132 public class Reservation implements Trip {
2133
2134     private Itinerary itinerary;
2135     private Passenger passenger;
2136
2137     /**
2138      * The Reservation constructor
2139      *
2140      * @param passenger the passenger on the trip
2141      * @param itinerary the itinerary for the trip
2142      */
2143     public Reservation(Passenger passenger, Itinerary itinerary) {
2144         this.passenger = passenger;
2145         this.itinerary = itinerary;
2146     }
2147
2148     /**
2149      * Gets the passenger for the trip
2150      *
2151      * @return the reservation's passenger
2152      */
2153     public Passenger getPassenger() {
2154         return passenger;
2155     }
2156
2157     /**
2158      * Gets the total airfare cost for the trip
2159      *
2160      * @return the total airfare for the trip
2161      */
2162     @Override
2163     public int getAirfare() {
2164         return itinerary.getAirfare();
2165     }
2166
2167     /**
2168      * Gets the origin of the trip
2169      *
2170      * @return the starting airport code
2171      */
2172     public String getOrigin() {
2173         return itinerary.getOrigin();
2174     }
2175
2176     /**

```



```

2177     * Gets the destination of the trip
2178     *
2179     * @return the final airport code
2180     */
2181     public String getDestination() {
2182         return itinerary.getDestination();
2183     }
2184
2185     public Itinerary getItinerary() {
2186         return itinerary;
2187     }
2188
2189     /**
2190     * The information from the reservation to be
2191     * displayed to the user
2192     *
2193     * @return the info to be shown to the user
2194     */
2195     public String getData() {
2196         return "\n" + itinerary.getData();
2197     }
2198
2199     /**
2200     * Converts the Reservation data into a string
2201     *
2202     * @return the reservation as a string
2203     */
2204     @Override
2205     public String toString() {
2206         return passenger.getName() + "," + itinerary.toString();
2207     }
2208
2209     /**
2210     * Checks to see if the reservation is equal to an object
2211     *
2212     * @param o the object being compared
2213     * @return if the two objects are equal
2214     */
2215     @Override
2216     public boolean equals(Object o) {
2217         if (o == null || getClass() != o.getClass())
2218             return false;
2219
2220         Reservation that = (Reservation) o;
2221
2222         if (this.getOrigin().equals(that.getOrigin())) {
2223             if (this.getDestination().equals(that.getDestination()))
2224                 return passenger.equals(that.passenger);
2225         }
2226         return false;
2227     }
2228
2229     /**
2230     * Creates a hash code for the object
2231     * This method was auto-generated by IntelliJ
2232     *
2233     * @return the hashCode of the reservation
2234     */
2235     @Override
2236     public int hashCode() {
2237         int result = itinerary.hashCode();
2238         result = 31 * result + passenger.hashCode();
2239         return result;
2240     }
2241 }
2242
2243 package Model.Components;
2244
2245 /**

```

```

2246     * This class is the Component object for the Composite
2247     * pattern. The methods inside are overridden by its
2248     * children.
2249     *
2250     * @author Matt Antantis
2251     * @author Ian Randman
2252     */
2253     public interface Trip {
2254
2255         /**
2256          * Gets the airfare cost for a trip
2257          *
2258          * @return the airfare cost
2259          */
2260         int getAirfare();
2261
2262         /**
2263          * Gets the origin of the trip
2264          *
2265          * @return the origin of the trip
2266          */
2267         String getOrigin();
2268
2269         /**
2270          * Gets the destination of the trip
2271          *
2272          * @return the destination of the trip
2273          */
2274         String getDestination();
2275
2276         /**
2277          * Gets the data to show to the user
2278          *
2279          * @return the data of the trip
2280          */
2281         String getData();
2282     }
2283
2284     package Model.Managers;
2285
2286     import Model.Components.Airport;
2287
2288     import java.io.FileNotFoundException;
2289     import java.io.FileReader;
2290     import java.util.HashMap;
2291     import java.util.Map;
2292     import java.util.Scanner;
2293
2294     /**
2295      * This class holds all the airports for the system
2296      * and responds to requests for information about them.
2297      *
2298      * @author Matt Antantis
2299      * @author Mark Vittozzi
2300      */
2301     public class AirportManager extends Manager {
2302         private Map<String, Airport> airports;
2303
2304         /**
2305          * The constructor for the AirportManger
2306          */
2307         public AirportManager() {
2308             airports = new HashMap<>();
2309
2310             try {
2311                 buildList();
2312                 buildConnections();
2313                 buildDelays();
2314                 buildWeather();

```

```

2315         } catch (FileNotFoundException e) {
2316             System.err.println(e.getMessage());
2317         }
2318     }
2319
2320 /**
2321  * Builds the list of Airports by reading a text file
2322  *
2323  * @throws FileNotFoundException throws an exception if cities.txt is not found
2324  */
2325 private void buildList() throws FileNotFoundException {
2326     reader = new Scanner(new FileReader("data/cities.txt"));
2327     System.out.print("initializing airports");
2328     while (reader.hasNext()) {
2329         String[] line = reader.nextLine().split(",");
2330         airports.put(line[0], new Airport(line[0], line[1]));
2331         System.out.print(".");
2332     }
2333     System.out.println("done");
2334 }
2335
2336 /**
2337  * Updates the airports in the list with the connection
2338  * time from a text file
2339  *
2340  * @throws FileNotFoundException if connection_times.txt is not found an exception
2341  * will be thrown
2342  */
2343 private void buildConnections() throws FileNotFoundException {
2344     reader = new Scanner(new FileReader("data/connection_times.txt"));
2345     System.out.print("initializing connections");
2346     reader.nextLine();
2347     while (reader.hasNext()) {
2348         String[] line = reader.nextLine().split(",");
2349         airports.get(line[0]).setConnectionTime(Integer.parseInt(line[1]));
2350         System.out.print(".");
2351     }
2352     System.out.println("done");
2353 }
2354
2355 /**
2356  * Updates the airports in the list with the delay
2357  * time from a text file
2358  *
2359  * @throws FileNotFoundException if delay_times.txt is not found an exception will
2360  * be thrown
2361  */
2362 private void buildDelays() throws FileNotFoundException {
2363     reader = new Scanner(new FileReader("data/delay_times.txt"));
2364     System.out.print("initializing delays");
2365     reader.nextLine();
2366     while (reader.hasNext()) {
2367         String[] line = reader.nextLine().split(",");
2368         airports.get(line[0]).setDelay(Integer.parseInt(line[1]));
2369         System.out.print(".");
2370     }
2371     System.out.println("done");
2372 }
2373
2374 /**
2375  * Updates the airports in the list with the weather data
2376  * from a text file
2377  *
2378  * @throws FileNotFoundException if weather.txt is not found
2379  */
2380 private void buildWeather() throws FileNotFoundException {
2381     reader = new Scanner(new FileReader("data/weather.txt"));
2382     System.out.print("initializing weather");

```

```

2382         while (reader.hasNext()) {
2383             String[] line = reader.nextLine().split(",");
2384             for (int i = 1; i < line.length; i += 2) {
2385                 airports.get(line[0]).addWeather(line[0] + "," + line[i] + "," + line[i]
2386                     + 1]);
2387             }
2388             System.out.print(".");
2389         }
2390         System.out.println("done");
2391     }
2392     /**
2393     * retrieves airport via airport code
2394     *
2395     * @param code code being used to retrieve airport
2396     * @return airport
2397     */
2398     public Airport getAirport(String code) {
2399         return airports.get(code);
2400     }
2401
2402     /**
2403     * retrieves delay via airport code
2404     *
2405     * @param code airport code for airport
2406     * @return delay time
2407     */
2408     int getDelay(String code) {
2409         return airports.get(code).getOverlay();
2410     }
2411
2412     public String getAirportInfo(String airportCode, int id) {
2413         return airports.get(airportCode).getWeather(id);
2414     }
2415
2416     public void switchClientModule(int id, String server) {
2417         for (String s : airports.keySet()) {
2418             airports.get(s).changeModule(id, server);
2419         }
2420     }
2421
2422 }
2423
2424 package Model.Managers;
2425
2426 import Model.Components.Flight;
2427 import Model.Components.Itinerary;
2428
2429 import java.io.FileNotFoundException;
2430 import java.io.FileReader;
2431 import java.util.ArrayList;
2432 import java.util.Scanner;
2433
2434 /**
2435 * This class handles interactions with the list of flights
2436 *
2437 * @author Matt Antantis
2438 */
2439 public class FlightManager extends Manager {
2440
2441     private ArrayList<Flight> flights;
2442     private AirportManager airportManager;
2443
2444     /**
2445     * The FlightManager constructor
2446     */
2447     public FlightManager(AirportManager airportManager) {
2448         flights = new ArrayList<>();
2449         this.airportManager = airportManager;

```

```

2450
2451     try {
2452         buildList();
2453     } catch (FileNotFoundException e) {
2454         e.printStackTrace();
2455     }
2456 }
2457
2458 /**
2459  * Builds each flight from a line in a text file
2460  *
2461  * @throws FileNotFoundException if flights.txt is not found
2462  */
2463 private void buildList() throws FileNotFoundException {
2464     reader = new Scanner(new FileReader("data/flights.txt"));
2465
2466     reader.nextLine();
2467     while (reader.hasNext()) {
2468         String[] line = reader.nextLine().split(",");
2469         flights.add(new Flight(line));
2470     }
2471     System.out.println("Build all flights");
2472 }
2473
2474 /**
2475  * Builds a list of potential flights from an origin and destination
2476  *
2477  * @param origin        the origin airport code
2478  * @param destination the destination airport code
2479  * @return a list of all potential flights combinations
2480  */
2481 public ArrayList<Itinerary> getPotentialItineraries(String origin, String
destination, int legs) {
2482     ArrayList<Flight> potentialFlights = getFlightsBetween(origin, destination);
2483
2484     // Gather all single leg itineraries
2485     ArrayList<Itinerary> potentialItineraries = new ArrayList<>();
2486     for (Flight f : potentialFlights) {
2487         potentialItineraries.add(new Itinerary(new Flight[]{f}));
2488     }
2489
2490     // Checks for possible multi flight itineraries
2491     if (legs > 0) {
2492         ArrayList<Flight> fromOrigin = new ArrayList<>();
2493         ArrayList<Flight> fromDestination = new ArrayList<>();
2494         for (Flight f : flights) {
2495             if (f.getOrigin().equals(origin) && !potentialFlights.contains(f)) {
2496                 fromOrigin.add(f);
2497             } else if (f.getDestination().equals(destination) &&
!potentialFlights.contains(f)) {
2498                 fromDestination.add(f);
2499             }
2500         }
2501
2502         // Checks to see if the two flights form a valid itinerary
2503         for (Flight ori : fromOrigin) {
2504             for (Flight desti : fromDestination) {
2505                 if (checkConnection(ori, desti))
2506                     potentialItineraries.add(new Itinerary(new Flight[]{ori,
desti}));
2507             }
2508         }
2509
2510         // Checks for extra valid itineraries if 2 connections are allowed
2511         if (legs > 1) {
2512             ArrayList<Flight> temp;
2513             for (Flight ori : fromOrigin) {
2514                 for (Flight desti : fromDestination) {
2515                     // Gets flights connecting the other two legs

```

```

2516         temp = getFlightsBetween(ori.getDestination(),
2517                                   desti.getOrigin());
2518
2519         // Checks to see if the three selected flights are valid
2520         for (Flight f : temp) {
2521             if (checkConnection(ori, f) && checkConnection(f, desti)) {
2522                 potentialItineraries.add(new Itinerary(new
2523                     Flight[]{ori, f, desti}));
2524             }
2525         }
2526     }
2527 }
2528 return potentialItineraries;
2529 }
2530
2531 /**
2532  * Builds a list of flights between two airports
2533  *
2534  * @param origin      the code of the origin airport
2535  * @param destination the code of the destination airport
2536  * @return the list of all flights between the two
2537  */
2538 private ArrayList<Flight> getFlightsBetween(String origin, String destination) {
2539     ArrayList<Flight> potential = new ArrayList<>();
2540     for (Flight f : flights) {
2541         if (f.validFlight(origin, destination))
2542             potential.add(f);
2543     }
2544     return potential;
2545 }
2546
2547 /**
2548  * Checks to see if two flights are able to connect
2549  *
2550  * @param first  the first flight
2551  * @param second the connecting flight
2552  * @return if the two flights can be connected
2553  */
2554 private boolean checkConnection(Flight first, Flight second) {
2555     if (first.getDestination().equals(second.getOrigin())) {
2556         int overlay = airportManager.getDelay(first.getDestination());
2557         if (overlay >= 60) {
2558             int hours = overlay / 60;
2559             overlay = overlay % 60;
2560             overlay += hours * 100;
2561         }
2562         return overlay + first.getNumericArrival() <= second.getNumericDeparture();
2563     } else
2564         return false;
2565 }
2566
2567 }
2568
2569 package Model.Managers;
2570
2571 import java.io.FileNotFoundException;
2572 import java.util.Scanner;
2573
2574 /**
2575  * Abstract class designed for creating
2576  * and maintaining a collection of objects
2577  *
2578  * @author Matt Antantis
2579  */
2580 abstract class Manager {
2581
2582     Scanner reader;

```

```

2583
2584 /**
2585  * Builds the collection of objects based off the implementation
2586  *
2587  * @throws FileNotFoundException if the file with the data is not found
2588  */
2589 private void buildList() throws FileNotFoundException {
2590 }
2591
2592 }
2593
2594 package Model.Managers;
2595
2596 import Model.Components.Flight;
2597 import Model.Components.Itinerary;
2598 import Model.Components.Passenger;
2599 import Model.Components.Reservation;
2600
2601 import java.io.*;
2602 import java.util.ArrayList;
2603 import java.util.Arrays;
2604 import java.util.Scanner;
2605
2606 /**
2607  * This class handles all the Reservations for the AFRS
2608  *
2609  * @author Matt Antantis
2610  */
2611 public class ReservationManager extends Manager {
2612
2613     private ArrayList<Reservation> reservations;
2614
2615     /**
2616      * The constructor for the ReservationManager
2617      */
2618     public ReservationManager() {
2619         reservations = new ArrayList<>();
2620
2621         try {
2622             buildList();
2623             System.out.println("..done");
2624         } catch (FileNotFoundException e) {
2625             System.out.println("Data not found");
2626         }
2627     }
2628 }
2629
2630 /**
2631  * Builds the reservations from a saved text file
2632  *
2633  * @throws FileNotFoundException if reservations.txt is not found
2634  */
2635 private void buildList() throws FileNotFoundException {
2636     System.out.print("restoring reservations\t");
2637     reader = new Scanner(new FileReader("data/reservations.txt"));
2638
2639     while (reader.hasNext()) {
2640         String[] line = reader.nextLine().split(",");
2641         Passenger tempPassenger = new Passenger(line[0]);
2642         Flight[] tempFlights = new Flight[Integer.parseInt(line[1])];
2643
2644         int index = 0;
2645         for (int i = 2; i < line.length; i += 6) {
2646             tempFlights[index] = (new Flight(Arrays.copyOfRange(line, i, i + 6)));
2647             index++;
2648         }
2649
2650         reservations.add(new Reservation(tempPassenger, new Itinerary(tempFlights)));
2651     }

```

```

2652     }
2653 }
2654
2655 /**
2656  * Adds a new Reservation to the reservations
2657  *
2658  * @param passenger the passenger for the reservation
2659  * @param itinerary the itinerary for the reservation
2660  * @return if the reservation was successful
2661  */
2662 public boolean addReservation(Passenger passenger, Itinerary itinerary) {
2663
2664     Reservation temp = new Reservation(passenger, itinerary);
2665
2666     if (reservations.contains(temp)) {
2667         return false;
2668     }
2669     reservations.add(temp);
2670     saveData();
2671     return true;
2672 }
2673
2674 /**
2675  * Removes a reservation from the list
2676  *
2677  * @param passenger the passenger to be removed
2678  * @param origin the reservation's starting location
2679  * @param destination the reservation's ending location
2680  * @return if the reservation was successfully removed
2681  */
2682 public boolean removeReservation(Passenger passenger, String origin, String
destination) {
2683     for (Reservation r : reservations) {
2684         if (passenger.equals(r.getPassenger())) {
2685             if (origin.equals(r.getOrigin()) &&
                destination.equals(r.getDestination())) {
2686                 reservations.remove(r);
2687                 saveData();
2688                 return true;
2689             }
2690         }
2691     }
2692     return false;
2693 }
2694
2695 /**
2696  * Gets the reservations for a passenger based on the origin and destination
2697  *
2698  * @param passenger the passenger with reservations
2699  * @param origin the passenger's origin
2700  * @param destination the passenger's destination
2701  * @return all the passenger's reservations
2702  */
2703 public ArrayList<Reservation> getReservations(Passenger passenger, String origin,
String destination) {
2704     ArrayList<Reservation> reserved = new ArrayList<>();
2705     if (origin.equals("")) {
2706         if (destination.equals("")) {
2707
2708             for (Reservation r : reservations) {
2709                 if (passenger.equals(r.getPassenger()))
2710                     reserved.add(r);
2711             }
2712         } else {
2713
2714             for (Reservation r : reservations) {
2715                 if (passenger.equals(r.getPassenger()) &&
                    destination.equals(r.getDestination()))
2716                     reserved.add(r);

```



```

2717         }
2718     }
2719     } else if (destination.equals("")) {
2720
2721         for (Reservation r : reservations) {
2722             if (passenger.equals(r.getPassenger()) && origin.equals(r.getOrigin()))
2723                 reserved.add(r);
2724         }
2725     } else {
2726
2727         for (Reservation r : reservations) {
2728             if (passenger.equals(r.getPassenger())) {
2729                 if (origin.equals(r.getOrigin()) &&
2730                     destination.equals(r.getDestination())) {
2731                     reserved.add(r);
2732                 }
2733             }
2734         }
2735     }
2736
2737     return reserved;
2738 }
2739
2740 /**
2741  * Saves the current reservations to a text file
2742  */
2743 private void saveData() {
2744     try {
2745         BufferedWriter writer = new BufferedWriter(new
2746             FileWriter("data/reservations.txt"));
2747         for (Reservation r : reservations) {
2748             writer.write(r.toString() + "\n");
2749         }
2750         writer.close();
2751     } catch (IOException e) {
2752         System.err.println(e.getMessage());
2753     }
2754 }
2755
2756 package Model.Managers;
2757
2758 import Model.Components.Weather.StoredWeather;
2759 import Model.Components.Weather.WeatherModule;
2760 import Model.Components.Weather.WebWeather;
2761
2762 import java.util.HashMap;
2763
2764 /**
2765  * An object that creates, holds, interacts with and modifies the weather modules
2766  * of a specific airport
2767  *
2768  * @author Mark Vittozzi
2769  */
2770
2771 public class WeatherManager extends Manager {
2772
2773     private HashMap<Integer, WeatherModule> weatherModuleMap;
2774     private HashMap<Integer, Integer> indexMap;
2775     private WeatherModule storedWeather;
2776     private String airportCode;
2777
2778     /**
2779      * @param airportCode: A string representing the name of the airport this object
2780      * belongs to
2781      */
2782     public WeatherManager(String airportCode) {

```

```

2783         this.weatherModuleMap = new HashMap<>();
2784         this.indexMap = new HashMap<>();
2785         this.storedWeather = new StoredWeather();
2786         this.airportCode = airportCode;
2787     }
2788
2789     /**
2790     * Adds the passed in weather to the module
2791     *
2792     * @param weather: A string representing weather
2793     */
2794     public void addWeather(String weather) {
2795         storedWeather.addWeather(weather);
2796     }
2797
2798     /**
2799     * Changes the weather module
2800     *
2801     * @param clientId: An int representing the user
2802     * @param server: a string representing the server to be switched to
2803     */
2804     public void switchModules(int clientId, String server) {
2805         if (server.equals("local")) {
2806             weatherModuleMap.replace(clientId, storedWeather);
2807         } else {
2808             weatherModuleMap.replace(clientId, new WebWeather(airportCode));
2809         }
2810     }
2811
2812     /**
2813     * Obtains the weather from the weather module
2814     * determines if the index variable is too large and resets it
2815     *
2816     * @param clientId: An int representing the users id
2817     * @return a string representing the weather for the airport
2818     */
2819     public String getWeather(int clientId) {
2820         if (!(weatherModuleMap.containsKey(clientId))) {
2821             indexMap.put(clientId, 0);
2822             weatherModuleMap.put(clientId, storedWeather);
2823         }
2824         if (weatherModuleMap.get(clientId).checkId(indexMap.get(clientId))) {
2825             indexMap.replace(clientId, 0);
2826         }
2827
2828         String weather =
2829             weatherModuleMap.get(clientId).getWeather(indexMap.get(clientId));
2830         indexMap.replace(clientId, indexMap.get(clientId) + 1);
2831
2832         return weather;
2833     }
2834
2835     /**
2836     * sets the delay of the storedWeather Module
2837     *
2838     * @param delay a string representing a delay
2839     */
2840     public void setDelay(String delay) {
2841         storedWeather.setDelay(delay);
2842     }
2843 }

```