

# Group Contribution

The members of the group consisted of Nadia Abulhawa, Haikah Ghoghari, Jamie Hill, Amy McFarland and Sam Muir.

In respect to the coding aspect of the group project, each member was assigned particular classes which was exhibited in the group planning schedule in the development report. Jamie was assigned the File IO and report generator classes, Nadia was assigned the flight classes, Haikah coded the passenger and baggage classes, Sam was assigned the Booking classes and Amy was designated the GUI classes. Throughout this part of the project, each group member assisted other members in their classes and testing which proved beneficial in completing the classes on schedule. Jamie offered assistance with the testing and flight classes and providing advice for other members classes. Nadia, Amy and Haikah worked together frequently helping each other complete their classes. Sam also provided assistance when it was required. Additionally, every group member read through each class that was committed to GitHub. This ensured that before each class was committed the group was happy that the classes included everything required. The integration of all the code was completed by \*\* which involved correcting any bugs that occurred in the program. After reading the report and ensuring the code worked effectively, the submission was done as a group to ensure everyone was satisfied with the work that had been accomplished.

As a group, we worked effectively together and often met to assist other members with their classes and discuss the objectives that had to be fulfilled for the next meeting. Each group member worked and contributed equally to the project.

In regard to the group report, each section was completed as a group with each member contributing equally. Amy wrote the summary and group contribution section of the report which was checked by the entire group. The class diagram restructuring was discussed as a group and individually completed by Nadia. The data structures section of the report was completed by Amy and checked by the group. The program functionality decisions were discussed a group during the coding part of the project and written by Amy. Finally, the testing section was completed by Sam. Before submission, the report was read by each group member to ensure everyone was satisfied with the completed work.

# Repository Link

https://github.com/jamiefhill/hw-ase-stage-one-check-in-system

# Status Report

This program meets the specification fully.

# Class Diagram

Activity Diagram (for calculating the excess baggage fees)

# Data Structures

The original data structures that were discussed in the development report have not changed and the decisions on which structures to use have remained the same. Though additional data structures were found that needed to be added

The booking collection class used a HashMap structure, for the same reasons that were provided in the development report. For example, it would allow the finding of a booking quickly for the user when their booking code was entered into the GUI. This proved beneficial and effective as it did not disrupt the flow of the program for the user.

The flight collection class implemented a TreeSet structure as it provided the optimum functionality for the final output report as it iterated through the set of flights. Additionally, it offers a sorted list of flights for the report.

# Program Functionality

As a group, when analysing the required functionality, we wanted to keep it simple and produce what the specification requested. When we looked at the requirements the first thing that stood out to us was the need for a booking collection and a flight collection. This is where we started in building our application. For the booking collection, as explained in the data structures section, we went for a HashMap data structure and wrapped this in a class to produce a BookingCollection class that would allow us to use class methods to interface the required functionality of the Data Structure. This meant we have a find method and a find per flight method to return booking objects that match the criteria. We took a similar approach for the Flight data structure and wrapped it in a class to produce a FlightCollection that allows us to find flights. Within both of these collections are the individual Booking and Flight object that allow us to manipulate the actual data for individual Bookings and Flights. From here we added a Passenger class to contain the Passenger information and the passenger class has a Baggage class that allows us to add baggage and calculate the excess costs for the passenger. The 2 collection classes have a load method that accesses a CSV processor class, this has a single responsibility to convert a string line with comma separated values into a String Array, which is added to an Array List that represents each line of the file. It also does the opposite to allow us to write CSV files back out to file. The CSV Processor class uses a FileIO class which reads in each line of a file to create an ArrayList for those lines and similarly takes an Array List and writes it out to a file. The load data methods for the Flights and Bookings are done within the constructor, taking only a String FileName parameter to load each one’s respective content. This is done before the GUI classes are created or displayed, meaning that we can pass a fully loaded Booking and Flight collection to the display part of our system. From here we can enter the Booking code, which follows a structure of two letters followed by a hyphen and then a three digit numbers such as **BA123-121** and also a last name from which we can look up a booking by code and then access a Passenger object to compare that the last names match. We then request that the passenger enter baggage length, width, height and weight before confirming the check in. If there is an excess baggage fees then this is displayed in a dialog box before going back to the main GUI screen for the next passenger to Check In. The excess charge was calculated by the allowed weight on a flight divided by the total possible passengers on a flight. If the passengers baggage weight minus the allowed weight per passenger was greater than zero then the amount greater than zero is multiplied by the excess charge value. These values are loaded from the Flight object that is looked up once a booking is found. We also needed a way to persist passengers check ins between the 2 GUI windows that we use, so the confirm gui and check in gui both take booking and flight collections as parameters to their constructors allowing any changes made to either collection to be passed back to the other window as the use of the application progresses. Finally we have to write out a report on the status of each flight when the application is closed. We do this by taking all the flight objects as a collection from the Flight Collection and use an iterator to work through each flight, then we request all bookings for that flight code from the Booking Collection findBookingsByFlightCode method and this allows us to generate the required statistics per flight before building a String Array, for columns, into an ArrayList, for rows, and passing that to the CSV processor to write out the required report.

# Testing

JUnit testing was completed for the