CSCI-605 Advanced Object-Oriented Programming Concepts

Homework 4

1 Introduction

We have often seen that there are many possible solutions to a particular problem, such as sorting or storing data in a list. These solutions may have different benefits and disadvantages compared to each other. We want to create a framework in which different solutions can be plugged in with minimal overhead and yet guaranteed to work.

In this homework will simulate an airline's passenger boarding system. Unlike other airlines, this airline does not preassign seats. Instead, each passenger checking in is assigned a boarding group (A-C) and a sequence number (1-60). The group and sequence number effectively represent a priority where group A passengers are allowed to board before passengers in group B; group B passengers board before group C. Passengers in the same group are prioritized by sequence number (1 is high; 60 is low).

When the plane is ready to board, passengers are instructed to line up according to group and sequence number. Upon entering the plane, passengers may select any available seat.

Clearly, it is advantageous to be in the "A" group; passengers in the "C" group are generally stuck with the center seats in the rear of the plane.

We can't guarantee the order in which passengers will arrive. However, the boarding queue must always reflect the correct order, even if all passengers have not yet arrived at the gate and entered the queue.

1.1 Provided Files

1. `hw4.HeapQueue` - This class is an implementation of a priority queue using a heap. **Note that initially most of the file is commented out**.
2. `sample_run.txt` A sample run showing the input and output of the simulation program.

1.2 The Simulation

The simulation of the boarding system will be implemented in a class called `Flight`. This class will be the entry point of the system and it will read the user input. It should prompt for one of three commands:

1. Add passenger to the boarding queue
2. Remove and print the next passenger from the queue with the highest priority
3. Quit

These commands should be entered as integers as given above. You must validate the input is a valid choice, and prints an error if not (do not terminate the program).

In the first command, you should ask for the name, group, and sequence number, in that order and add the new passenger to the queue.

The second choice simulates the boarding process where the next passenger in the queue is allowed to actually board the plane. This passenger should be the highest priority passenger from your queue. If the queue is empty, your program should handle this gracefully as in the sample run.

In the third case, your program should end.

2 Implementation

2.1 Part 1 - A: The PriorityQueue Interface

You must define a `PriorityQueue` interface in such way that you do not have to make any modifications to the provided `HeapQueue.java` file.

This interface must define all the methods required to add and remove passengers into the queue. It must also provide a method to check is the queue is empty.

Later in this course, you will learn how to define generic interfaces. This is beneficial in that we can reuse implementations for other problems that require this sort of queue. But for this homework, you are not required to implement a generic interface.

2.2 Part 1 - B: The Comparable Interface

We need to ensure that passengers are boarding based on their priority. We will determine that priority through the Java interface Comparable.

The `Comparable` interface defines a single method `compareTo` that returns an integer value indicating if "this" (the current object) is less than, greater than, or equal to a second object.

In this case, we will be determining the "order" or boarding "priority" based on the boarding group and sequence number. If Joe is in boarding group A, he will board before Sue in boarding group B. If Sally and Sam are both in boarding group A, we need to look at the sequence number to determine who boards first. In short - you will determine the "priority" of a passenger based on the boarding group and sequence number.

Write a `Passenger` class that, in addition to storing the relevant data, also implements the `Comparable` interface.

2.3 Part 2 - Another PriorityQueue implementation

The priority queue solution that we provided (a heap) is fine in most circumstances, but if passengers generally arrive with increasing priority (i.e. each new passenger will go to the front of the line), a list-based solution will likely be more efficient. In this part of the homework, you will write an implementation of a priority queue that uses linked nodes (call this class `LinkedQueue`). This should be based on the interface you have implemented for Part 1 - A. When this is complete, you should be able to run your original solution to Part 1 without any changes at all other than the type of PriorityQueue that you create!

3 Testing

In order to receive the full credit for testing, you will need to write a unit test in the `hw4.test` folder for `LinkedQueue`, `Passenger` and any extra classes that you have implemented in your solution. You do not need to write a unit test for the standalone execution of the `Flight` class.

4 Submission

You will need to submit all of your code, including your tests, to the MyCourses assignment before the due date. You must submit your `hw4` and `hw4.test` folders as a ZIP archive named "`hw4.zip`" (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

5 Grading

The grade breakdown for this homework is as follows:

- Explanation: 80%
- Implementation: 75%
  - Simulation (Flight) 15%
  - Part 1 - A (PriorityQueue) 5%
  - Part 1 - B (Passenger) 20%
  - Part 2 (LinkedQueue) 35%
- Testing: 15%
- Style: 10%