CSCI-605 Advanced Object-Oriented Programming Concepts

Homework 5: Studio 54

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, we are interested in keeping track of the line of patrons anxious to get into the famous Studio 54. As gatekeeper, we get to choose which patrons to admit. As they arrive, we put them in line somewhere, keeping in mind that we would like to admit patrons with a higher "coolness" factor before those with lesser "coolness". Among equally cool folk, we want to admit regulars before non-regulars. A space in the club may open up (and be filled by the coolest person in line) at any time.

1. What sort of data structure would you use for this problem?
2. What methods would such a structure need?
3. Can we make this a generic data structure? Would there be any limitations on what sort of objects could be put in such a data structure? If so, what are they?
4. When testing such a data structure, what sort of test cases might be useful? Remember that we haven't decided how it might be implemented yet!
5. Now consider the patrons themselves. We need to put them in line as mentioned above. What state should be included in the Patron class?
6. What methods (behaviors) should be included?

1.1 Goals

Students will gain experience with the following:

- Interfaces
- Generic Types
- Bounded Parameter Types

1.2 Provided Files

1. `HeapQueue.java` - This class is an implementation of a priority queue using a max-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.

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 be a generic type, i.e. it can be specified for different data types. This is beneficial in that we can reuse implementations for other problems that require this sort of queue. However, not just any object can be used - in order to do its job, the queue has to be able to determine the priority of any object that is being added.

Thus, we need to ensure that regardless of the type of object being put in the queue, we can determine its priority so we know where to position it in the queue. We will do this through the Java interface Comparable.

2.2 Part 1 - B: The Simulation

The simulation of the gatekeeper's job will be implemented in a class called `Gatekeeper`. 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 a patron to the queue
2. Admit the highest priority patron
3. Close for the night (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, coolness factor (integer value in the range 1-10), and regularity (input as y/n, and converted to boolean), in that order and add the new patron to the queue.

The second choice removes a patron (this will be the highest priority patron!) from the queue and print his or her details. If the queue is empty, your program should handle this gracefully as in the sample run.

In the third case, your program should end.

Your output does not need to exactly match the given example, but keep it clean and easy to follow.

2.3 Part 1 - C: The Comparable Interface

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 "priority" based on their coolness factor and regularity. If Jennifer Aniston has a coolness factor of 8, she will enter before John Cusack that has a coolness factor of 3. If Ryan Gosling and Emma Stone have the same coolness factor, we need to look if they are regulars to determine who enter first. If two patrons have the same coolness factor and regularity, either of them can enter first.

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

2.4 Part 2 - Another PriorityQueue implementation

The priority queue solution that we provided (a max-heap) is fine in most circumstances, but if patrons generally arrive with increasing priority (i.e. each new patron 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!

NOTE: to receive credit for this portion of the lab, you must build your `LinkedQueue` from scratch. That is, **you may not use any built-in Java classes related to linked data structures or priority queues**.

3 Testing

In order to receive the full credit for testing, you will need to write a unit test in the `test` folder for `LinkedQueue`, `Patron` 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 `Gatekeeper` 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 source code and `test` folders as a ZIP archive named "`hw5.zip`" (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

5 Grading

The following deductions will apply:

- Completeness/Implementation: up to 100% if a part of the writeup is not observed.
- Correctness/Testing: up to 50% or affected part if the solution is not correct.
- Quality/Style/Documentation: up to 50% if solution is poorly designed.
- Explanation: up to 100% if the student is not able to respond grader's questions.