CSE214

HOMEWORK - SUMMER 2016

HOMEWORK 3 - due Tuesday, August 2nd no later than 7:00PM

REMINDERS:

- **Be sure your code follows the coding style for CSE214.**
- **Make sure you read the warnings about academic dishonesty. *Remember, all work you submit for homework or exams MUST be your own work.***
- **Login to your grading account and click "Submit Assignment" to upload and submit your assignment.**
- **You may use any predefined Java Development Kit (JDK) data structure classes such as ArrayList, Vector, or LinkedList or Queue.**
- **You may use Scanner, InputStreamReader, or any other class that you wish for keyboard input or reading a text file.**

The purpose of this assignment is to perform simulation of a building's elevator system that processes passenger requests. You will need to input parameters to the simulation representing the number of floors in the building, the number of elevators, the simulation length in time units, and the percent chance that a passenger will place a new elevator request during each time unit. The following list of details further describes how the simulation will be run:

- **There will be one queue of requests for the simulation. The queue is a standard FIFO queue, meaning that the first request placed is the first to be granted.**
- **Each elevator can only handle one passenger at a time.**
- **A single time unit represents the amount of time it takes for the elevator to move up or down one floor.**
- **At the beginning of each time unit, there is a random chance that a new request will be placed by a passenger. There may be at most one new request placed per time unit. Requests have a randomly generated source floor (where the passenger is) and destination floor (where the passenger wishes to go).**
- **After checking for a possible new request, all elevators that are currently idle (i.e., not handling a request) will be given a request to handle, if such a request exists. It does not matter which elevator is assigned to handle a request.**
- **At the end of each time unit, all elevators that are processing requests move one floor closer to completing the request. If the elevator was moving to pick up a passenger, the elevator will move one floor closer to the source floor. If the elevator was moving to drop off the passenger, it will move one floor closer to the destination floor.**
- **If the source floor is reached, the elevator will begin heading towards the destination floor on the next time unit.**
- **If the destination floor is reached, the elevator will then be marked as idle (on the next time step, it can handle a new request).**
- **If an elevator just begins handling a request on the current time unit and if it was already on the source floor, it may begin moving towards the destination on the current time step.**
- **If an elevator is on the source floor and if the source floor is also the destination floor, the elevator may be set to idle (it has reached its destination and can handle a new request on the next time unit).**
- **Your task is to calculate the average waiting time for requests.**
  Wait time is defined as the number of time units that pass from when a request is placed on the queue until the elevator picks up the passenger (i.e., until the elevator arrives at the source floor).
  You will add the amount of waiting time for a request to the waiting sum and increment the request count as each request has its elevator reach its source floor.
  In your calculations, you must ignore all requests that have not had an assigned elevator reach its source floor.

  Note: When an elevator has reached the source floor of the request it is handling, the waiting time officially ends. The time it takes the elevator to reach the destination is not counted.

The following are the class specifications that you will be required to follow for your implementation of this simulation:

1. Write a class called Request that should contain the following member variables:

- sourceFloor (int)
- destinationFloor (int)
- timeEntered (int - the time that this request was placed on the queue)

You should also provide for this class the following methods:

- **A constructor that takes as a parameter the number of floors in the building.**
  The two integers represent the values for sourceFloor and destinationFloor, which will be randomly generated within this constructor.
  The random values must be between 1 and the number of floors in the building, inclusive.
  The timeEntered will be undefined at first (the mutator will set this).
- **Accessor and mutator methods for each variable.**

Hint: To generate the random sourceFloor and destinationFloor, use the Math.random() method.

2. Write a fully-documented class named RequestQueue that must be derived as a subclass of Vector (or other class) from the Java API. You should provide a constructor and the standard methods enqueue(), dequeue(), size(), and isEmpty() as were discussed in lecture.

3. Write a fully-documented class named BooleanSource that is the same as the BooleanSource object discussed in lecture. This class has a double named probability as a member variable, a constructor that accepts a double as a parameter as the value of this member variable, and also a boolean method called requestArrived() that returns true a percentage of the time equal to probability (and otherwise it returns false).

Hint: To determine the return value of the requestArrived() method, you should use the Math.random() method.

4. Write a class called Elevator that should contain the following member variables:

- **currentFloor (int)**
- **elevatorState (an int constant, either IDLE, TO_SOURCE, or TO_DESTINATION)**
- **request (Request object representing the request being handled or null if the Elevator is idle)**

You should also provide for this class the following:

- **A default constructor that sets request to null, elevatorState to IDLE, and currentFloor to 1.**
- **Accessor and mutator methods for each variable.**
- **Final variables to represent IDLE, TO_SOURCE, and TO_DESTINATION.**

Hint: The values of IDLE, TO_SOURCE, and TO_DESTINATION do not matter as long as they are distinct.

5. Write a fully-documented class named Simulator will be used to actually carry out the simulation. This class will contain a single static method called simulate that accepts the following four parameters, carries out the simulation, and prints the results:

- **The probability of a request being introduced per time unit (a double between 0.0 and 1.0, inclusive)**
- **The number of floors in the building (an int greater than 1)**
- **The number of elevators in the building (an int greater than 0)**
- **The length of the simulation in time units (an int greater than 0)**

6. Write a fully-documented class named Analyzer containing a main method which prompts the user, on separate lines, for each of the 4 parameters required for the simulate method of the Simulator class. Make sure to check that each of the values for these parameters are within the valid range and then run the actual simulation.

7. Include any exception classes that you will be using, if any, for your simulation.

Note: You may also include any additional classes that you wish to use.

INPUT FORMAT:

- **The ranges for the parameters are given above and they must be checked for errors.**
- **Each parameter is to be entered after a prompt on a separate line.**

OUTPUT FORMAT:

- **You should display the average waiting times for handling requests.**
- **All averages should be rounded to two decimal places.**
- **Keep in mind that the simulation results should be random and will vary from one running of the simulation to another.**
- **Read the section below about how you should test your program to make sure that your program works correctly.**

TESTING YOUR PROGRAM

How do you know if your program is working properly?

1. **Test your program using a very SHORT simulation first.**
2. **Include output statements in your program to print out when a request arrives in a queue, which elevator is selected, and when it reaches the source floor. Based on this output, you can compute the average waiting times by hand probably and compare with the program results.**
3. **However, you don't really want the print statements in a simulation that runs for a large number of commands. Why bother erasing all those debugging statements? Here's a way to solve the problem. Put your output statements in an if statement:**
   ```
   if (debug) {
       // print debugging information here
   }
   ```
   You will need a local boolean variable "debug" which you can set to true while you are testing your program. Once you have tested it fully, you can set debug to false instead, recompile your program, and run it on longer simulations. No need to erase that code!
4. **Make a menu command that toggles on and off a tracing mode, to be stored in a boolean variable similar to debug in the previous example. Combine this with print statements wrapped in if statements to give yourself detailed information about every step of the simulation that can be disabled at any time you wish.**

NOTE: All of these testing suggestions are highly recommended, but they are NOT mandatory for the completion of your assignment.

SAMPLE INPUT/OUTPUT:

```
Note that computer output is in blue and what might actually be occurring in the simulation is in green.
Also, points at which calculating variables must be incremented are in red.
Keep in mind that this is just a random occurrence and the results that your simulation obtains will vary.

Welcome to the Elevator simulator!

Please enter the probability of arrival for Requests: 0.3
Please enter the number of floors: 5
Please enter the number of elevators: 2
Please enter the length of the simulation (in time units): 11

//Now, the actual simulation will occur, although this does not need to be printed.
//To show the queue, we will use the notation (source, destination, time entered) to show requests
//The fronts of the queue is to the left and the rear is to the right
//Elevators have the notation [Floor X, state, request_being handled]

Step 1: A request arrives from Floor 2 to Floor 5
         //The request is immediately handled, so it is placed on the queue and immediately taken off.
   //The first elevator was idle, so it starts moving towards the source floor
         Requests:
         Elevators: [Floor 2, TO_SOURCE, (2, 5, 1)], [Floor 1, IDLE, ---]
Step 2: Nothing arrives
         //The elevator reached the source floor, so the waiting is over for the first passenger.
   //The first elevator now moves towards the fifth floor (the destination)
   //At this point we know that the waiting time for the first request is 1 time step (2-1=1)
         Total Wait Time = 1, Total Requests = 1
         Requests:
         Elevators: [Floor 3, TO_DESTINATION, (2, 5, 1)], [Floor 1, IDLE, ---]
Step 3: A request arrives from Floor 4 to Floor 1
         //The request is immediately handled, so it is placed on the queue and immediately taken off.
   //The second elevator was idle, so it starts moving towards the source floor
         Requests:
         Elevators: [Floor 4, TO_DESTINATION, (2, 5, 1)], [Floor 2, TO_SOURCE, (4, 1, 3)]
Step 4: A request arrives from Floor 5 to Floor 1
   //The request is placed on the queue.
   //The first elevator reaches its destination, so it is now idle and can take a request next time unit.
         Requests: (5, 1, 4)
         Elevators: [Floor 5, IDLE, ---], [Floor 3, TO_SOURCE, (4, 1, 3)]
Step 5: Nothing Arrives
   //The request on the queue is to be handled by the first elevator.
   //The second elevator reaches its source, so heads for its destination starting on the next time step.
   //The first elevator is already on the source floor, so it gets its passenger and moves towards the destination.
   //Since both passengers were picked up, the waiting time variables may be updated.
         Total Wait Time = 4, Total Requests = 3
         Requests:
         Elevators: [Floor 4, TO_DESTINATION, (5, 1, 4)], [Floor 4, TO_DESTINATION, (4, 1, 3)]
Step 6: Nothing Arrives
         Requests:
         Elevators: [Floor 3, TO_DESTINATION, (5, 1, 4)], [Floor 3, TO_DESTINATION, (4, 1, 3)]
Step 7: Nothing Arrives
         Requests:
         Elevators: [Floor 2, TO_DESTINATION, (5, 1, 4)], [Floor 2, TO_DESTINATION, (4, 1, 3)]
Step 8: Nothing Arrives
         Requests:
```

```
         Elevators: [Floor 1, IDLE, ---], [Floor 1, IDLE, ---]
Step 9: A request arrives from Floor 1 to Floor 3
         //The request is immediately handled, so it is placed on the queue and immediately taken off.
   //The first elevator was idle, and it was already on the first floor, so it moves towards the destination.
   //We can update the wait time variables now.
         Total Wait Time = 4, Total Requests = 4
         Requests:
         Elevators: [Floor 2, TO_DESTINATION, (1, 3, 9)], [Floor 1, IDLE, ---]
Step 10: A request arrives from Floor 4 to Floor 3
         //The request is immediately handled, so it is placed on the queue and immediately taken off.
   //The first elevator was idle, so it moves towards the source.
         Requests:
         Elevators: [Floor 3, IDLE, ---], [Floor 2, TO_SOURCE, (4, 3, 10)]
Step 11: Nothing Arrives
         Requests:
         Elevators: [Floor 3, IDLE, ---], [Floor 3, TO_SOURCE, (4, 3, 10)]
//At the end of simulation, the second elevator didn't yet reach the request source floor so its wait time wasn't included.

//END OF SIMULATION
//Total Wait Time = 4
//Total Requests = 4

//Now for the actual program output:

Total Wait Time: 4
Total Requests: 4
Average Wait Time: 1.00
```

EXTRA CREDIT (Optional - 15 points) :

NOTE: It is important that you do not modify any of your classes from parts 1-7 while working on the extra credit. You are still required to submit the mandatory parts and they must work as described above to achieve fully credit for this assignment. A working extra credit part is not a substitute for a working mandatory part.

The elevator algorithm provided above is not an optimal solution because it only allows a single passenger to be in the elevator at any given time and because each elevator can only handle a single request at once. For the extra credit part of the assignment, you may add the following additional classes to your submission that optimize the simulation:

8. Write a fully-documented class named OptimalSimulator will be used to actually carry out the simulation. This class will contain a single static method called simulate that accepts the following four parameters, carries out the simulation, and prints the results:

- **The probability of a request being introduced per time unit (a double between 0.0 and 1.0, inclusive)**
- **The number of floors in the building (an int greater than 1)**
- **The number of elevators in the building (an int greater than 0)**
- **The length of the simulation in time units (an int greater than 0)**

Replace the following assumptions by the simulation rules described above:

- **Elevators now also have a direction, up or down. As an elevator passes a floor, all waiting passengers are picked up that want to go in that direction.**
- **An elevator moves until it no longer has any requests (sources or destinations) in the direction it was moving. At this point, it is now idle. In the documentation of this class describe what algorithm is being used to service the requests.**
- **Elevators can carry an unlimited number of passengers.**
- **Idle elevators can move in any direction to pick up requests.**

You may keep track of elevator directions in an additional boolean array (true = up, false = down). You may use any additional data structures necessary to keep track of requests that have not yet been handled (such as a list for requests on each floor).

9. Update the main method of the Analyzer class so it asks the user which method (regular or optimal) must be run. In both cases, you must check the input parameters given to you by the user. Make it clear what option (yes/no, 0/1, etc.) runs which algorithm (regular or optimal).

Course Info | Schedule | Sections | Announcements | Homework | Exams | Help/FAQ | Grades | HOME