**ICS 22 / CSE 22 Fall 2012  
Project #4: *Gone to the Movies***

**Due date and time:** *Monday, November 19, 11:59pm*

*This project is to be completed individually*

***Background***

In business or scientific contexts, it's sometimes very costly to make changes to the way that you're doing things, yet sometimes the changes you consider might yield great benefits; but sometimes, you don't know if they'll yield those benefits without some experimentation. But if the experimentation is too costly, you might never be willing to pay the cost of finding out whether the changes you're considering are worthwhile or not.

Computers have a role to play in situations like this. If you can write a program that will show you how things would be different after you make your changes, and compare that result to how things are now, you can more cheaply find the answer you're looking for — should I make this change or not? — and, if it turns out to be a positive change, you can proceed with making it happen.

In this project, we'll consider a situation like this one. Suppose that the movie theater chain Millennium Cinemas is interested in finding ways to attract customers away from their larger, more well-established rivals. With the proliferation of "megaplex" movie theaters during the last couple of decades, Millennium management believes that one problem that movie-goers face is the long wait in line to buy tickets. They seek a solution that will allow their customers to spend less time in line, hoping that this will dramatically improve their sales. It's not in their best interest to spend a lot of time running experiments, especially since some of the things they might want to try could well end up lengthening their customers' waits in line, ultimately hurting their business. (When you're a large company, you can spend money trying things that might have a negative impact, absorb the loss, and move on. When you're small, you don't have this kind of freedom.)

So, to aid in their analysis of the time that customers spend in line, you'll build a program that will simulate various arrangements of lines and ticket windows, to allow them to find the optimal combinations for its theaters. The project will give you practice implementing queues, as well as introduce you to the concept of a *simulation*, a program that approximates a real-world situation (in this case, the movement of customers as they arrive at the theater, stand in line, and buy tickets) and keeps track of statistics about it. You'll also have a chance to gain some design skills, as you'll be starting this project from scratch. Don't worry, though; we'll give you plenty of help along the way.

***Reminder: Do not partner up***

For this project, your work is expected to be completed individually, so *do not* partner up, and *do not* use the pair programming technique.

***The program***

Millennium requires a program that will allow them to simulate a variety of arrangements of ticket lines, ticket windows, and patterns of customer arrival. You will write a program that gives them this flexible simulation ability. It will allow the user to specify the number of ticket windows, whether there will be a single line or one per window, and the speed at which each ticket attendant can process customers. Once these simulation parameters have been specified, the program processes a file that indicates the arrival of customers, simulating their arrival into the ticket lines, tracking how long each customer spends in line, and ultimately calculating statistics about them.

***Window and line arrangements***

Millennium has many different styles of theaters, from single-screen theaters showing classic or foreign films to larger multi-screen varieties showing the latest releases. To support its various sizes and styles of theaters, it wishes to investigate two arrangements of ticket windows and lines:

1. One or more ticket windows, each with its own line.
2. One or more ticket windows, and one line feeding all of them.

For each arrangement of lines, they will also be interested in investigating the number of windows necessary to support various numbers and patterns of customers. They will use the simulator to determine the contexts in which each arrangement will be appropriate.

***The input file***

In order to simulate different types of theaters, the simulator will need to accept a set of parameters, specified in an input file. The input file specifies the number of ticket windows, and whether there is a single ticket line or a separate line for each window. In addition, the input file will contain a list of *customer arrivals*, specifying how many customers will arrive at the theater at which times. Here is an example of an input file. The italicized portions are included here for descriptive purposes, and should not be included in the actual input file.

Short simulation *brief description of the simulation*

5 *the length of the simulation, in minutes*

2 *the number of ticket windows*

S *how many lines: "S" for single, "M" for multiple*

45 *number of seconds it takes to process a customer at window #1*

35 *number of seconds it takes to process a customer at window #2*

1 30 *one customer arrives 30 seconds into the simulation*

5 35 *five customers arrive 35 seconds into the simulation*

3 45 *three customers arrive 45 seconds into the simulation*

1 60 *one customer arrives 60 seconds into the simulation*

1 90 *one customer arrives 90 seconds into the simulation*

END *the "END" tag marks the end of the customer arrivals*

The input file is broken into two parts: the *setup section*, which describes the arrangement of ticket windows and lines, and the *customer arrival section*, which lists the arrival of customers.

**The setup section**

The first line of the setup section is a brief description, perhaps a sentence or so, that explains the purpose of the simulation. The next line specifies the length of the simulation. Following that, the next line specifies how many ticket windows there will be during the simulation. This is followed by either "S" or "M" on a line by itself, which specifies whether there is a single ticket line or one ticket line per window. Finally, there is a line for each of the windows — they should be numbered from 1..*n* — that states how many seconds it takes the attendant at that window to sell a ticket to a single customer.

There must be a positive number (i.e., greater than zero) of ticket windows, though there is no pre-defined maximum. The number of seconds that each attendant takes to process a customer must also be a positive number.

After reading the setup section, the simulator should be ready to begin processing the arrival and departure of customers, so any applicable data structures should have been created and initialized appropriately.

**The customer arrival section**

Each line in the customer arrival section of the input file describes the arrival of a number of customers at a particular time. The time is specified as the number of seconds since the simulation started, and they must increase as the file is processed (i.e., the time on each line in the file must be greater than the time on the line that came before it). The number of customers must be positive.

When customers arrive, they get into one of the ticket lines. If there is only a single line, obviously they get into that line. If there are multiple lines, each customer should be processed one by one; each one should get into the line which has the fewest number of customers already in it. In the event of a tie, they should always get into the one corresponding to the lowest-numbered window. There is no maximum number of customers that can be in a particular line.

When a window is unoccupied, a customer *immediately* moves from its corresponding ticket line to the window. That customer will stay for the appropriate number of seconds. At that time, the customer will leave the window and will *immediately* be replaced by another.

For the sake of simplicity, we'll assume for our simulator that customers may not move from one ticket line to another. In other words, suppose there is one ticket line per window. If there is no one at the first window and no one in the first line, but there are three customers in the second line, we will assume that the customers in the second line *will not* go to the first window. (Obviously, this is a non-issue when there is only one single line.)

**Assumptions about the input file**

You may assume that the input file is formatted as described above. You may *not* assume that it will be exactly the file that we're showing as an example, but you may assume that there won't be a word where a number is expected, and that there will always be exactly two integers on a line that's supposed to have two integers. Further, you may assume that other restrictions, such as customer arrivals being in ascending order with respect to time, will hold.

***How will you know how to run my program?***

The name of the input file should be specified as a command-line argument, as in [Project #3](http://www.ics.uci.edu/~thornton/ics22/LabManual/Simple). If you'd like to refresh your memory about how to do that in Java (and within Eclipse), read through the relevant section of the [Project #3](http://www.ics.uci.edu/~thornton/ics22/LabManual/Simple)write-up.

The class with your main() method should be called Simulator, so we'll be able to run your program in a predictable way.

***The final report***

When the simulation is complete, your simulator should print out the *final report*. The final report prints out some summary statistics about the simulation to **System.out**. The final report must contain the following information:

* For each ticket window:
  + The number of tickets sold at that window. You may assume that each customer buys exactly one ticket.
  + The percentage of time spent idle. You can calculate this by counting the number of seconds when no customer was at that window, and divide it by the total number of seconds in the simulation.
* For each ticket line:
  + The number of customers still waiting in that line.
  + The maximum length that line ever reached.
  + The average amount of time customers waited in that line to buy tickets. Count only the customers who reached a window, though you may count them if they are still at the window when the simulation ends.
  + The maximum amount of time any customer waited in that line, including customers who are still in line at the time the simulation ends.
* Overall:
  + The total number of tickets sold at all windows.
  + The average amount of time customers waited in line to buy tickets. Count only the customers who reached a window, though you may count them if they are still at the window when the simulation ends.
  + The average amount of time each customer spent at a ticket window. Naturally, you should only count customers who reached a window and bought a ticket, so don't count customers who are still at the window when the simulation ends.

***How should the example file be processed?***

What follows is a step-by-step explanation of the processing of the example file shown in the previous section. At the conclusion of the example, the statistics are shown. All times are reported in terms of seconds. The final output of your program does not need to look like this, but I do suggest that you include something like this in your program's output; there is simply no way you'll get the final statistics to come out correctly if customers aren't moving to the right places at the right times during the simulation. (This also provides us the ability to give you partial credit if you're not able to get the entire program finished; as problems get more complex, it's worth considering ways to reach stable ground and have partial solutions working, and this partial output is a great example of that.)

* time 0: window #1 and window #2 are idle, no one is in line
* time 30: one customer arrives and gets into line
* time 30: customer goes to window #1 (spent 0 seconds in line)
* time 35: five customers arrive and get into line
* time 35: customer goes to window #2 (spent 0 seconds in line)
* time 45: three customers arrive and get into line
* time 60: one customer arrives and gets into line
* time 70: customer leaves window #2 (spent 35 seconds at window #2)
* time 70: customer goes to window #2 (spent 35 seconds in line)
* time 75: customer leaves window #1 (spent 45 seconds at window #1)
* time 75: customer goes to window #1 (spent 40 seconds in line)
* time 90: one customer arrives and gets into line
* time 105: customer leaves window #2 (spent 35 seconds at window #2)
* time 105: customer goes to window #2 (spent 70 seconds in line)
* time 120: customer leaves window #1 (spent 45 seconds at window #1)
* time 120: customer goes to window #1 (spent 85 seconds in line)
* time 140: customer leaves window #2 (spent 35 seconds at window #2)
* time 140: customer goes to window #2 (spent 95 seconds in line)
* time 165: customer leaves window #1 (spent 45 seconds at window #1)
* time 165: customer goes to window #1 (spent 120 seconds in line)
* time 175: customer leaves window #2 (spent 35 seconds at window #2)
* time 175: customer goes to window #2 (spent 130 seconds in line)
* time 210: customer leaves window #1 (spent 45 seconds at window #1)
* time 210: customer leaves window #2 (spent 35 seconds at window #2)
* time 210: customer goes to window #1 (spent 150 seconds in line)
* time 210: customer goes to window #2 (spent 120 seconds in line)
* time 245: customer leaves window #2 (spent 35 seconds at window #2)
* time 245: window #2 is idle
* time 255: customer leaves window #1 (spent 45 seconds at window #1)
* time 255: window #1 is idle
* time 300: simulation ends

***The statistics for the example file***

These are the statistics that you should see after processing the example file. Note that the decimal numbers may be off by a hundredth in some cases for you, depending on how (and whether) you handle rounding, and that's fine.

* Window #1
  + 5 tickets sold
  + 25.00% of time was idle (75 seconds out of 300)
* Window #2
  + 6 tickets sold
  + 30.00% of time was idle (90 seconds out of 300)
* Line
  + No customers were waiting at the end of the simulation
  + Maximum length was 8 customers
  + Avg. wait time: 76.82 seconds
  + Maximum wait time: 150 seconds
* Overall
  + 11 tickets sold
  + Avg. wait time: 76.82 seconds
  + Avg. time spent at a window: 39.55 seconds

***How about some additional examples?***

To test your simulation, you'll want to run more examples than just the one example file that I provided. However, I'm only going to be providing you with the one example. Part of the goal of writing a program is to find a way to be sure that it works; that means that you need to test it. In the case of a simulation, that means you need to come up with some interesting inputs, figure out for yourself what the output should be, then see if your program generates the right output. That's part of what I'm expecting you to do for this project, though you will not be required to submit your additional input files. In fact, because it can be an arduous task to build input files and figure out what their expected output is, I encourage you to share your input files and expected outputs with one another, so that everyone can benefit from one another's insights about testing the program.

***Some suggestions (and requirements) about the design of your simulator***

There are many ways that you could implement your simulator, and I will not impose any particular one on you, though I will make a few specific requirements. For the most part, you're on your own for this project. However, here are some suggestions which might help you get started, along with a few design requirements.

**Deciding on your classes**

A somewhat simplistic, but often effective, way to break a large design into classes is to consider a description of the desired system and "look for the nouns." While this technique won't give you a complete or perfect design, it will at least give you a good feel for what the major abstractions in your program are. Using that approach for this program, you might discover some of the following ideas (and probably others). Of course, depending on how you describe your program, you may discover different abstractions than these.

* *Customer*. The simulator is intended to keep track of customers as they enter lines, enter windows, and leave the area with their tickets.
* *Queue*. A queue is an object that manages a set of objects standing in line waiting for access to some shared resource. In the case of this program, the objects are customers and the shared resource is a ticket window; however, the queue class should be built to be generic, so that it could be reused in other programs.
* *Ticket line*. One of the central features of the simulation. Customers enter at the back of lines, work their way up, then emerge from the front. This is an example of the concept of a queue, which you will be required to use to implement your ticket line. A good way to implement your ticket line is to have it contain a queue. The distinction between a queue and a ticket line is that a queue is generic (i.e., it can store any kind of object) and is intended to be possibly reused in other programs, while a ticket line specifically manages customers and keeps tracks of statistics necessary for this program.
* *Ticket window*. Another central feature of the simulation. One customer at a time can occupy a ticket window. For each window, there is a predetermined amount of time that each customer will need to spend there. It is possible for a window to be "idle" (i.e., to have no customer in it). A ticket window will need to keep track of the necessary statistics that are specific to that window.
* *Theater*. A broader abstraction for the theater area. Contains ticket lines and ticket windows. The purpose of the theater abstraction is to prevent other parts of the program from needing to be aware of the arrangement of lines and windows. For example, when a customer arrival is processed, the Theater object can be in charge of placing the customers in the appropriate lines. The Theater object should also track summary statistics for the entire simulation.
* *Simulation*. An object that manages the entire simulation, delegating tasks to the theater (which will delegate some tasks to other objects, such as ticket lines and ticket windows). This is where the input file can be read and processed. This object will also manage the *simulation loop*, which I'll describe a bit later.
* *Clock*. Since the simulation has a clock that keeps track of the current time, expressed as the number of seconds since the simulation started, it makes sense to have one object, shared by all of these classes, that keeps track of the current time.

**Using a queue to implement your ticket line**

You are required to build a queue class for this project and use it to implement your ticket line. The queue should be generic (e.g., Queue<E>), so that it could be used in other programs. It is required to be implemented as a linked list, with the major operations (enqueue, dequeue, and front) written so that they will run in *O*(1) time.

Separately, you'll probably find the need to define a ticket line class that specifically deals with customers and keeps track of the statistics that are necessary for this program. (There are two reasons why it's wise to separate the queue from the ticket line: so that the queue can be reused in other programs and so that you can separate the logic that stores and manipulates the queue of customers from the logic that tracks the statistics.)

**The simulation**

Your program should load the setup information from the input file first, and then setup the simulation (i.e., create the theater object with the appropriate initial state) based on those values. From there, you can implement the simulation in various ways, but I suggest the following approach. (This is by no means the most efficient way to implement such a simulation, but is efficient enough for our purposes, striking a good balance between efficiency and ease of implementation for this course.)

Write a *simulation loop*, each iteration of which simulates one second. Each second, the loop performs the following major tasks:

* Find out what the current time is. Remember that the current time is tracked as the number of seconds since the simulation started. (Obviously, this should start at zero.)
* If new customers are arriving in line at the current time, add them to the appropriate lines.
* Check each ticket window. If there is no customer there, the first customer in its corresponding ticket line should be removed from line and arrive at the window. If there is a customer there, see if the customer is finished. The customer is finished if she has been at the window for the specified number of seconds. If the customer is finished, she leaves, and should be immediately replaced by the first customer in the corresponding ticket line (if any).

When the specified number of seconds of simulation time elapse, the simulation ends, and the accumulated statistics should be printed out.

**Memory usage**

You *may not* load all of the customer arrivals into memory at the beginning of the simulation. Instead, you should read the customer arrivals into memory one line at a time, when necessary. Reading all of the customer arrivals into memory ahead of time will yield a substantial penalty on your **Quality of solution** score for this project.

***Testing***

To satisfy the testing portion of this project, write a JUnit-based test class that tests your Queue implementation. It is not necessary to test other parts of your program, but you're welcome to test anything else you wish using JUnit.

One thing to understand about unit testing is that how you design your program has a big effect on whether unit tests are easy (or even possible) to write. This is a positive feedback loop: a design that permits unit tests to be written is one that is usually, on balance, better than the alternative. Consider ways that you can make as much of your code unit-testable as possible.

***Limitations***

You must implement your own linked list functionality. You may not use pre-existing linked list implementations, such as **java.util.LinkedList** in your solution. If you'd like, you may reuse your LinkedList<E> class from previous projects, though you will need to make it a singly-linked list with both head and tail references, so that the various queue operations will run in constant time; you may also need to add one or more methods to it.

***Starting point***

Unlike previous projects, you will not be provided with any code as a starting point for this one. I believe that you now have the skills to write this program from scratch. There is quite a bit of design advice in the write-up above, so be sure to spend some time reading and understanding it. Ask questions in discussion and chat on the newsgroup. Plenty of help is available, but I encourage you to start early.

***Deliverables***

You must submit all of your **.java** files. Please do not turn in the **.class** files, or other files generated by your development environment. Follow [this link](http://www.ics.uci.edu/~thornton/ics22/LabManual/SubmittingProjects.html) for a discussion of how to submit your project.

* Testing section adjusted by Alex Thornton, Winter 2012.
* Section detailing use of JUnit 4 in Eclipse rewritten (as user interface had changed) by Alex Thornton, Fall 2008.
* Testing requirement restored and rewritten by Alex Thornton, Spring 2007.
* Design advice rewritten, testing requirement removed, and some of the other requirements adjusted (to fit with previous projects in this quarter's ICS 22 offering) by Alex Thornton, Fall 2004.
* More minor changes by Alex Thornton, Fall 2002.
* A few more modifications and the inclusion of *jUnit* into this project by Alex Thornton, Spring 2002.
* Re-tooled for the "new" ICS 22 (including the addition of a testing component) by Alex Thornton, Winter 2002.
* A few screws were tightened by Alex Thornton, Summer 2001.
* Some minor fixes and disambiguation by Alex Thornton, Spring 2001.
* Originally written by Alex Thornton, Winter 2001.