CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Part 3]

Preparation

Assume the students:

- have already attempted Lab 1G
- have an understanding of the customer/server system being simulated
- have already encapsulated the given code into their appropriate classes

Objective

After completing the lab, students should:

- be familiar with interface and abstract class
- be able to implements from interface or extends from abstract class
- appreciate programming to interface
- be ready for *Lab 2G*

Goals

There is a new `Main.java` file uploaded to deal with changing input/output. The new `Main.java` will **NOT** be compatible with the previous `Simulator.java` file. The new function signature for `simulate` is as follows with the changes described below.

```
public static String simulate(int[] custIDs, double[] custArriveTime, int numArrival, int servMaxCust, int numServer, int servQueueSize)
```

- The function accept an additional `integer`, `numServer` to indicate the number of servers
- The function accept an additional `integer`, `servQueueSize` to indicate the number of slots in the waiting queue

Grading

This lab is **ungraded** but will be extended to *Lab 2G*. As stated before, the following grading will be used for *Lab 2G* (*with possible changes of at most* $\pm 5\%$):

- **Correctness:** 40%
  - This will be enforced automatically. If there are n number of test cases and you can solve m of them correctly, then you will get $\frac{m}{n} \times 40$. However, there will be *private* and *hidden* test cases.
- **Design:** 60%
  - *Modularity/Functional Abstraction:* 10%
    - Penalty will be given for code that are supposed to be abstracted into another functions.
    - These include duplicated codes, codes that are too complex (e.g., too long or too nested). You should aim such that your function can be visible in its entirety in a single screen.
  - *Encapsulation:* 10%
    - Penalty will be given for wrong access modifiers (e.g., using `public`, `default`, or `protected` for fields that are supposed to be `private`).

- Penalty will be given for non-`public` access modifier with unrestricted accessor and mutator (i.e., `public` methods that can easily access and modify the non-`public` field without any checks).

- *Data Abstraction:* 30%
  - Penalty will be given for data that are not abstracted into classes.
  - You may want to consider *noun* in your creation of classes.
- *Miscellaneous Constraints:* 10%
  - Penalty is given for non-conformance of constraints.
  - This may include:
    - Use of specific interface: we may require you to create specific interface and implement them.
    - Use of specific inheritance, final keyword, etc.

Discrete Event Simulator

A discrete event simulator is a software that simulates a system (often modeled after the real world) with events and states. An event occurs at a particular time, each event alters the state of the system, and may generate more events. A discrete event simulator can be used to study many complex real-world systems. The term *discrete* refers to the fact that the states remain unchanged between two events and therefore the simulator can *jump* from the time of one event to another instead of following the clock in real time. The simulator typically keeps track of some statistics to measure the performance of the system.

In this lab, we will simulate the following situation (*any changes will be highlighted in yellow with explanation on the changes in italics and underlined*):

- We have a shop with **one type** of *server*.
  - Each server has a unique ID. The first server has the ID = 0.
  - There are at most 1000 servers.
  - The shop has **multiple** server.
    - *There can be up to* `numServer` *servers where* `numServer` ≥ 1.
    - *The first server has the ID = 0. Any subsequent server s has the ID = $s - 1$.*
    - *We will need to log the server in our logging system too in the following format:*
      - *When a customer is served, we log which server is serving the customer*
        - *Example:* `0.500 2 served by 2`
      - *When a customer is waiting, we log which server the customer is waiting for*
        - *Example:* `0.500 2 waits for 1`
      - *When a customer is done, we log which server was serving the customer*
        - *Example:* `1.500 0 done with 0`
  - The server can serve **multiple** customer at a time.
    - The server will not be busy until it is currently serving the maximum number of customers it can handle.
    - For instance, if `servMaxCust` variable above is 3, then the server can serve up to 3 customers at a time, after which the server marks itself busy until one of the customer is done.
  - The server takes **the same amount** of service time to serve.
  - The server will **not take a rest**.

  - Each server has their own waiting queue for customer to wait.
- Our shop accepts **one type** of *customer*.
  - Each customer has a unique ID. The ID is a running number based on the arrival order of the customer. The first customer has the ID = 0.
  - There are at most 1000 customers.
  - When the customer arrives, the customer will look for *non-busy* server (i.e., may serve additional customer).
  - If there is no *non-busy* server, the customer will join a waiting queue of the server **with the lowest ID** if there is a space.
  - If there is no space on any waiting queue, the customer will **leave and not come back at a later time**.
  - If the customer is in a waiting queue, the customer will **wait until served and not leave**.
- We have exactly **one type** of *waiting queue* for customers.
  - Customer waiting in a waiting queue will be **ordered based on arrival time**.
  - There is **multiple** slot in the waiting queue.
    - *The number of slot is* `servQueueSize` *where* `servQueueSize` ≥ 1.
    - *All servers will have the same number of slots.*
- We have **five** *events* happening in the shop.
  - There are at most 100 events in event queue at any point in time. There can be more than 100 events in total, but if you are removing older events before adding new events (*excluding arrival*), you can be sure that the number of events are 100 or fewer.
  - The five events are:
    - Customer arrive (`ARRIVES`)
    - Customer waiting (`WAITS`)
    - Customer served (`SERVED`)
    - Customer leaves (`LEAVES`)
    - Customer done (`DONE`)
  - If two events are happening at the same time, the events must be different event. For instance:
    - No two customers can arrive at the same time.
    - No two customers can be served/done served at the same time.
      - **CLARIFICATIONS:**
      - The statement "*at the same time*" means the same *logical time*.
      - Consider a set of events happening at the same time as follows:
        - [<DONE, 2.0>; <ARRIVES, 1.5>; <DONE, 1.5>]
        - *There is NO same event happening at the same time*
        - The event <DONE, 1.5> is processed first based on specification above
          - It may introduce new event <SERVED, 1.5>
        - [<DONE, 2.0>; <ARRIVES, 1.5>; <SERVED, 1.5>]
        - *There is NO same event happening at the same time*
        - The event <SERVED, 1.5> is processed first based on specification above
          - It may introduce new event <DONE, 2.5>

- [<DONE, 2.0>; <ARRIVES, 1.5>; <DONE, 2.5>]
- *There is NO same event happening at the same time*
- The event <ARRIVES, 1.5> is processed first based on specification above
  - It may NOT introduce new event <SERVED, 1.5>
    - Because introducing <SERVED, 1.5> will lead to <DONE, 2.5>
    - This will eventually lead to [<DONE, 2.0>; <DONE, 2.5>; <DONE, 2.5>]
    - *There is same event happening at the same time in this case which is marked in RED*
  - It may introduce new event <WAITS, 1.5>
    - Because we don't know yet when that customer will be served, it will eventually lead to [<DONE, 2.0>; <DONE, 2.5>]
    - *There is NO same event happening at the same time in this case*

- The events are ordered in an increasing order of priority. For instance:
  - If customer is done being served occurs at exactly the same time as another customer arrives, we will execute the customer done being served first before processing the arrival of another customer.
  - If this leads to yet another customer being served, then we serve this yet another customer before processing the customer that has just arrived, and thus may be freeing a slot in the waiting queue.

In the specification above (*and one below* later), any phrases in **bold** are subject to change. Besides the specification above, we have a description of events as follows.

- When a customer *arrives*:
  - If there is a server that can still serve a customer, the customer will go to the server with the lowest ID.
  - If all servers are busy, the customer will join a waiting queue that is not yet full based on the customer's *preference above*.
  - If all waiting queue are full, the customer will leave.
- When a customer *waits* in a waiting queue:
  - The customer may or may not leave based on the customer's *preference above*.
- When a customer is being *served* by a server:
  - Nothing of interest happens.
- When a customer *leaves*:
  - The same customer (with the given ID) may return after a time based on the customer's *preference above*.
- When a customer is *done* being served by a server:
  - If there is a customer in the waiting queue of the server, the server will introduce an event of serving the customer in the queue.

Furthermore, we are interesting in the following statistics about the simulation. Given a sequence of customer arrivals where **the time of each arrival is given**[1]:

- What is the average waiting time for customers that have been served?
- How many customers are served?
- How many customers left without being served?

In your Lab 1U, you are given a simple discrete event simulator to answer the questions above. The code given is in a purely imperative style. Furthermore, it is badly written as all the codes are written in the `simulate` method, without any modularity, abstractions, or encapsulation. The only good thing is that input/output is already done for you in `Main.java` so you don't have to care about it for now. You are to study the code given and rewrite and/or refactor the code to OO style by creating your own classes. Based on the description above, the recommendation is to have at least three classes: `Server`, `Customer`, and `Event`. Additionally, you may want to have a class `Simulator` to perform the main process.

Input/Output

Due to the changing requirement, it is unavoidable that input/output may change. However, the changes will be minimal and until we learn more about Input/Output, the `Main.java` will be given that contains the input reading and the final output formatting. You are not to change the file `Main.java` as well keep the function signature for `simulate` in `Simulator.java`. The input format is as follows:

- The first line consists of three integer: `servMaxCust`, `numServer`, and `servQueueSize`.
- Then there is an arbitrary number of inputs. You are to read until there are no more input to read.
- Each line after the first is a single `double` value t. t is the time of arrival of the next customer.
    - t is always an increasing sequence.

[1] **BIG NOTE:** this is in bold, and this specification may be changed.