CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Part 5]

Preparation

Assume the students:

- have already attempted Lab 2G

Objective

After completing the lab, students should:

- be able to generate random number
- be ready for *Lab 3G*

Goals

You will no longer be given `Main.java` and/or `Simulator.java` and you should write all your files on your own instead.

Grading

This lab is **ungraded** but will be extended to *Lab 3G*. As stated before, the following grading will be used for *Lab 3G* (*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:* 10%
    - 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:* 30%
    - 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.

CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Part 5]

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 **two 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).
  - Customer #1: General 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 come back at a later time**. The time is specified as `custReTime` and will be different for each customer.

        - If the customer is in a waiting queue, the customer will **wait until served and not leave**.
    - Customer #2: Elderly 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**.
        - The server will serve the elderly first before serving the General Customer until there are no more elderly in the queue.
    - Additional Constraints:
        - You are to name the classes as `GeneralCustomer` and `ElderlyCustomer`.
- We have exactly **one type** of *waiting queue* for customers.
    - Customer waiting in a waiting queue will be **ordered based on arrival time as well as type**.
        - Elderly Customer will be prioritized over General Customer.
    - 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.
    - Additional Constraints:
        - You are to use only one queue for both General Customer and Elderly Customer.
        - You are NOT allowed to have two different queues.
- We have **five** *events* happening in the shop.
    - There are at most 100 events in event queue at <u>any point in time</u>. 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.
- *The next customer being served should be from Elderly Customer first before General Customer.*

Furthermore, we are interesting in the following statistics about the simulation. Given a sequence of customer arrivals where **the time of each arrival as well as customer type is randomly generated**[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?

The random generation of customer arrival time is done using exponential random variable generator. We call our generator `CustomerGenerator` class. The `CustomerGenerator` class has to implement the following functions:

```
public CustomerGenerator(long seed, double probElderly,
                         double arrivalRate, double rearriveRate);

public ICustomer nextCustomer();
```

In the constructor, `seed` is the seed to be used in the random number generator; `probElderly` is the probability that a given customer is an elderly customer; `arrivalRate` is the arrival rate of the customer; and `rearriveRate` is the re-arrival rate of general customer who leaves.

The probability that a customer is an elderly customer should be a *uniform distribution* with probability `probElderly`. The class should have exactly two `java.util.Random` classes as its attributes

```
1. typeRand = new Random(seed);     // RNG for choosing customer type
2. custRand = new Random(seed + 1); // RNG for customer attributes
```

You may use the following functions to generate exponential random variable:

```
private double rng(Random rand, double rate) {
  return -Math.log(rand.nextDouble())/rate;
}
```

The method `nextCustomer()` should not call `nextDouble` unless necessary. This will prevent different values from being generated on subsequent calls. In particular, generating `ElderlyCustomer` requires 1 call to `typeRand.nextDouble()` and 1 call to `custRand.nextDouble()` while generating `GeneralCustomer` requires 1 call to `typeRandNextDouble()` and 2 calls to `custRand.nextDouble()`.

For conformity, we enforce the following pseudo-code to generate the `ICustomer`:

1. Generate a random `double` between `[0.0, 1.0)` using `typeRand`.
2. If the generated `double` is *less than or equal* to `probElderly` then go to step 3. Otherwise go to step 4.
3. Create an `ElderlyCustomer` object by generating the arrival time. You have to call `custRand.nextDouble()` once here or if you are using `rng` method above, you call `rng` once by giving it `custRand` so that `custRand.nextDouble()` is invoked exactly once.
    - Use the random value generated to *calculate* the arrival time of this customer
    - Then *return* the newly created `ElderlyCustomer` object
4. Create a `GeneralCustomer` object by generating the arrival time and the re-arrival time. You have to call `custRand.nextDouble()` twice here or if you are using `rng` method above, you call `rng` twice by giving it `custRand` so that `custRand.nextDouble()` is invoked exactly once.

- Use the ***first*** random value generated to ***calculate*** the arrival time of this customer
- Use the ***second*** random value generated as the re-arrival time for this customer.
- Then ***return*** the newly created `GeneralCustomer` object

Note

If your `ElderlyCustomer` and/or `GeneralCustomer` constructor does not accept the arrival time, then you need to first generate all the required `double` values at the constructor for `CustomerGenerator` class ***in the correct order as specified in the pseudocode***. The `nextCustomer` method above will then use these pre-generated `double` values to instantiate the `ICustomer` object when needed.

However, this means that every `double` value must be generated in the order given above.

Input/Output

The input format is as follows:

- The first line consists of one `integer`: `numCustomer`.
  - This indicates the number of customers we are going to simulate.
  - $1 \leq$ `numCustomer` ≤ 1000
- The second line consists of three `integer`: `servMaxCust`, `numServer`, and `servQueueSize`.
- The third line consists of one `integer` and three `double`: `seed`, `probElderly`, `arrivalRate`, and `rearriveRate`.

---

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