CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Initial Design]

Preparation

Assume the students are already familiar with:

- basic Java syntax: class, fields, methods, functions
- basic Java concepts: data abstraction, functional abstraction, encapsulation
- writing Java program using IDE of choice
- compiling and running Java programs

Objective

After completing the lab, students should:

- be more comfortable with basic Java syntax and semantics
- be more comfortable in creating Java classes from scratch
- be more comfortable with abstraction barrier and modularizing complex problem
- be comfortable with following a given coding convention
- appreciate OOP in comparison to pure imperative paradigm

Goals

You are given a ***working*** discrete event simulator written without any abstraction in imperative style. Furthermore, no encapsulation is performed. The entire code is written in the `Simulator.java` file under the `simulate` method with the exception of certain constants written outside the `simulate` method. The discrete event simulator will simulate customers being served by servers.

The goal of this lab is to ***rewrite*** the simulator in OO style. You will need to encapsulate and create abstraction barrier to the various variables and methods. Here are some hints:

- Think about the problem you are solving: what are the nouns? Nouns are good candidates for classes.
- For each class, what are the attributes/properties relevant to the class? These are good candidates for fields in the class.
- For each class, what are their responsibilities? What can they *do*? These are good candidates for methods in the class.
- How do the objects of each class interact? These are good candidates for `public` methods. If a method is not intended to be `public`, you should declare them as `private`.

This lab is part 1 of 8 labs where you will build an increasingly complex discrete event simulator. Each subsequent parts will modify existing ones with possible additional constraints on design. You should aim to design your simulator in such a way that changes can be done in a minimal way. The principle we repeated during lecture is "*Each significant piece of functionality in a program should be implemented in just one place in the source code*". To that end, you can ask yourself the following questions:

- Is there any repeated code? Repeated code with minimal changes are good candidates for functional abstraction. Note that you may "*generalize*" the code before abstracting.
- Can the code structure (e.g., while-loop, switch-case, if-else, function) be seen in its entirety in a single screen? Codes that are too long makes reasoning about the program harder. This is another good candidate for functional abstraction.

CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Initial Design]

As we will slowly evolve the simulator into something more general and will simulate a more complex customers and/or server behaviours in our simulation. Making sure your code will be able to adapt to new problem statement is the key. Trying to solve the lab without considering this (e.g., *coming up a with solution that computes the average waiting time in only tens of lines of code*) will likely make your changes in subsequent labs harder as you will need to rewrite much of your solutions to handle the new requirement.

Grading

This lab is **ungraded** but will be extended to *Lab 1G* which will be graded. Coming up with good encapsulation will make your *Lab 1G* much easier to solve. To prepare for *Lab 1G*, the following grading will be used for *Lab 1G* (*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.

CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Initial Design]

In this lab, we will start by simulating a specific situation:

- 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 **one** server.
  - The server can serve **one** customer at a time.
  - 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 customers).
  - 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**.
- 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.
  - 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.

CS2030 Programming Methodology II

Semester 4 2018/2019

Discrete Event Simulator [Initial Design]

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`. Any changes in `Simulator.java` will be explained and you are expected to add the changes to your code. The input format is as follows:

- There is an arbitrary number of inputs. You are to read until there are no more input to read.
- Each line 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.