School of Information and Physical Sciences

SENG2200 – Programming Languages & Paradigms

Assignment 3 (15%)

Due Date: 11:59 pm on 07 Jun (Week 13)

1 Objectives

This assignment aims to test your understanding of OO design and Java programming. A successful outcome should be able to demonstrate a solution of the assignment tasks with correct Java implementation and report.

2 Design Problem

You will build a **discrete event simulation** (https://en.wikipedia.org/wiki/Discrete-event_simulation) to model a **widget production line**. This activity will be useful for future courses.

Note:
Although this problem seems to be solvable with concurrent programming, you are **NOT** supposed to use this approach.

3 Design Specifications

The Java program built should contain the components and conform to the specifications stated from Sections 3.1 to 3.9.

3.1 Notations

Table I Assignment 3 Parameters

Parameter	Description
Qmax	**Capacity** of inter-stage storages.
M	**Average processing time** of a widget in a stage, given as a program input.
N	**Range of processing time** in a stage, given as a program input.
P	**Processing time** of a widget.
T1	**Starting time** of widget production.
T2	**Completion time** of widget production.
d	A **random number** in range 0 to 1.

Your assignment will use the parameters listed in Table I.

3.2 Widget Production Line

You will simulate the production of "*widgets*" on a production line as illustrated in Fig. 1. The production line consists of a number of production **stages**, separated by **inter-stage storage** in the form of queues of finite length (size *Qmax*).

The inter-stage storages are necessary because the time taken to process a widget at any production stage will vary due to random factors. The production line will be balanced in that the average time taken at any stage will be effectively equal.

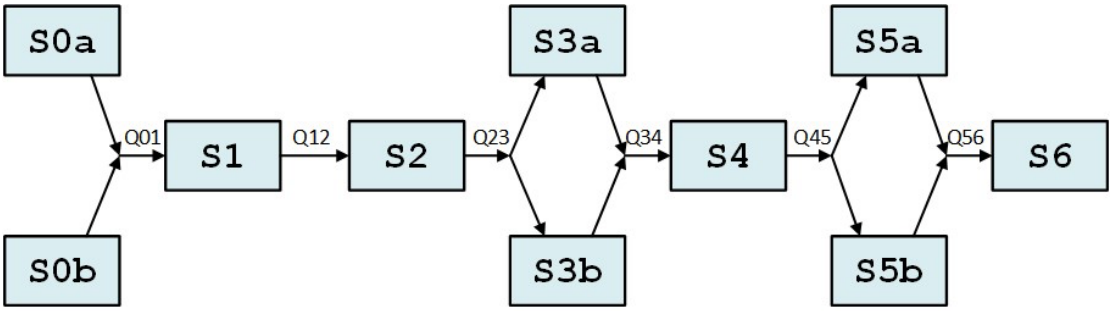

Fig. 2. The production line simulated in Assignment 3.

`S0a` and `S0b` are the **beginning stages**, while `S6` is the **final stage**. Stages `S3a/b` and `S5a/b` are **parallel stages** and share entry and exit storage queues. Q01 is the **inter-stage storage** between `S0a/b` and `S1`, and Q12 is the **inter-stage storage** between `S1` and `S2`, and so on.

Production at any stage for this program involves the following steps:

1. Take a widget from the preceding inter-stage storage.
2. Calculate how long it will take to process through this stage (using a random number generator).
3. After that amount of time, place the widget into the following inter-stage storage.

The production line parameters need to follow the following specifications:

- Stages `S0a` (excluding `S0b`), `S3a/b`, and `S5a/b` will have mean and range values of *2M* and *2N*, respectively; whereas all other stages have the mean and range values of *M* and *N*, respectively.
- The capacity of the inter-stage storages (*Qmax*) will always be greater than 1.
- The simulation **time limit** will be 10,000,000 time units ~~so that the production line will produce about 10000 widgets~~.

Note:

- Parallel stages must not be combined into a single object. You should produce an object for every **stage** and **inter-stage storage**. These objects should link together as shown in Fig. 2.
- You may not use an array or linear store to hold your **stages** and **queues**, i.e., you have to create Stage and Queue objects.
- You should not hardcode the values of *M*, *N*, and *Qmax* as they are arguments during program execution.

3.3 Widgets

A widget could be anything from cars to mousetraps. For this assignment, a widget will simply be an object capable of storing simulation time values of the important events throughout its production, such as time entering, and leaving, each production stage. This in turn allows us to calculate things like the average time waiting in each queue, etc.

A widget will be created in stages `S0a` and `S0b`. Each widget should include a unique ID (unique serial number) as a string, with the last letter being either '*A*' or '*B*' according to the beginning stage (i.e., 'A' for `S0a` and 'B' for `S0b`). As an example, you may generate a unique string *x* and the widget's unique ID could be either "*xA*" or "*xB*" depending on which stage (`S0a` or `S0b`) creates the widget.

The initial "*x*" part of the unique ID will be generated by a **Singleton** class (refer *Singleton Creational Design Pattern*) which provides a **`getID()`** method that shall return a unique string. After this, during stages S0a and S0b, the character '*A*' or '*B*' will be appended to this value. The final concatenation will then be stored as the widget's unique ID.

3.4 Simulation Time

In this simulation, we are interested in the amount of idle time at each station, the tame taken for a normal widget to be produced, and the overall throughput of the production line.

You will be implementing a **simple discrete event simulation** which can be controlled by time events (aka '*jobs*' that contain a comparable '*time*' the event will happen, and reference to '*location*' that the event will occur) that are placed into a *priority queue*. It is recommended to use the standard Java container **`PriorityQueue`** as it allows you to load comparable jobs into it to manage time.

The simulation time will start at **zero** and proceed until the **time limit** of the simulation. Time is best stored as a **double**. This greatly reduces the chance of two time values being equal within 13 significant figures. However, if two time values are in fact equal, you may assume first come, first served (FCFS).

3.4.1 Simulation of Random Processing Time

Given mean M and range N, the processing time of a widget is calculated as $P = M + N \times (d - 0.5)$.

In a stage, if a widget processing starts at $T1$, then you will know that this stage completes production on this widget at time $T2 = T1 + P$.

Note that d should be generated according to a **uniform probability distribution**. This is easily catered by using the standard Java random number generator from **`java.util`**. First, you can set up a random number generator r with

```
Random r = new Random();
```

Next, you can obtain the pseudo-random number d (in the range 0 to 1) from r using

```
double d = r.nextDouble();
```

3.5 Blocking and Starving

If a stage finish processing it's widget but finds the destination storage to be full, the stage must ***block*** until its successor stage takes a widget from the storage. This allows the stage to re-submit its widget to the storage and continue with the next widget.

If a stage wishes to proceed to produce its next widget but finds the predecessor storage empty, then it must ***starve*** until its predecessor stage completes production and places that widget into the intervening storage for further progress.

The beginning stage(s) of the production line is considered to have an infinite supply of (raw) widgets and hence, can never starve. The final stage is considered to have an infinite sized warehouse following and hence, can never be blocked. The simulation begins with all stages (except the beginning stages) in a starved state and all the inter-stage storages empty.

3.6 Unblocking and Unstarving

When a stage completes processing of a widget, it must check whether stages ***either*** side of it, which are currently ***blocked*** or ***starved***, may now be able to resume production.

For example, assume that stage `S1` is previously blocked because the storage Q12 following it is full. Once stage `S2` starts processing on a new widget, a widget would have been retrieved from storage Q12 (indicating a free space). Upon completing the processing of the new widget, ~~stage `S2` should check if stage `S1` is unblocked~~ stage `S1` should be checked and unblocked. The same logic also applies to unstarving.

3.7 Programming Language

Your program should be coded in **Java**. It is preferred to target the long term support version of Java currently installed on the University labs, i.e., **Java 17.0 (LTS)**.

4 Output Generation

The program should **produce statistics** on **each stage** which includes:

- **actual production** (as a percentage),
- how much **time** is spent **starving**,
- how much **time** is spent **blocked**.

For the **inter-stage storages**, calculate:

- the **average time** a widget spends in each queue,
- the **average number** of widgets in the queue at any time (this statistic will require some thought).

You will also keep a **total number of widgets** created by stages `S0a` and `S0b` that arrive at the end of the line, i.e., `S6`. Lastly, you will keep a total of the number of widgets that followed each path through stage `S3a/b` and stage `S5a/b`, that arrive at the end of the line.

All output is to the standard output. A sample output ~~of the input in Section 4.1~~ is as follows:

```
Production Stations:
-------------------------------------------
Stage:  Work[%]      Starve[t]      Block[t]
S0a      47.96%           0.00    103,601.39
S0b      48.65%           0.00    101,127.90
S1       99.87%       4,672.64     48,409.24
S2       98.87%      94,264.24     18,346.11
S3a      72.40%   2,236,949.88    522,533.24
S3b      63.02%   2,943,353.53    754,801.25
S4       98.61%     136,951.46      2,206.19
S5a      62.96%   2,957,251.09    746,649.33
S5b      72.32%   2,226,450.41    540,988.14
S6       98.82%     118,226.16          0.00

Storage Queues:
-----------------------------
Store   AvgTime[t]    AvgWdgts
Q01       3,674.47        3.12
Q12       3,734.88        3.11
Q23         515.32        0.40
Q34       4,085.28        3.39
Q45         414.71        0.29
Q56       4,011.52        3.31

Production Paths:
-----------------
S3a -> S5a:  2,296
S3a -> S5b:  1,321
S3b -> S5a:  3,973
```

```
S3b -> S5b:  2,272

Production Widgets:
-------------------
S0a:  2,296
S0b:  1,321
```

Note:

- **Do not deviate from the output form!** The output should be presented as above.
- The values in the sample are made up and should not be considered as correct values.
- Usage of random numbers will result in different values for you, but there is an expected range into which your results should fit. Deviation from these expected ranges will result in a loss of marks.

5 Submission of Deliverables

You are required to submit a **zip** file containing the executable code and report. The submission should be named according to the convention cXXXXXXX.zip, where cXXXXXXX is your student ID. If you submit more than once, then only the latest will be graded.

5.1 Executable Code

Classes are to be implemented and submitted in separate files, i.e., do not save all your classes into one source file, or write classes within classes (except where required by ~~**`Iterable`**, or if you choose to write your **`Node`** as an inner class~~ convention of implemented solution).
Name your start-up/entry-point class A3.java (capital-A number-3). This allows the marker to compile your program with the command:

```
javac A3.java
```

and to run your program with the command:

```
java A3 M N Qmax
```

within a Command Prompt window.

Note:

- Your program should be able to handle different arguments for testing, i.e., different values of *M*, *N* and *Qmax*.
- If your program cannot be compiled and executed (incl. run-time errors) by using the above commands, marks will be deducted.

5.2 Report

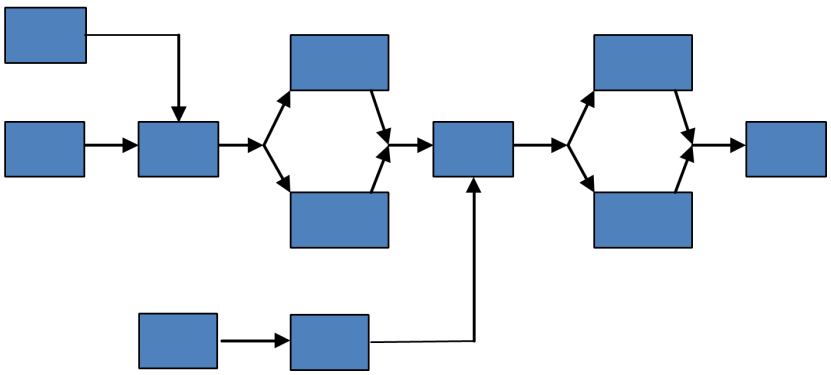

Fig. 1. Alternative widget production line for Assignment 3 report.

In conjunction with your program, present a **5 to 7 page** report which contains the following:

- Introduction
- Produce a UML class diagram that shows the classes (and interfaces) in your program and the relationship(s) between them.
- Comment on your use of inheritance and polymorphism and how you arrived at the particular inheritance/polymorphic relationships you used in your program.
- How easy will it be to alter your program to cater for a production line with a different topology, e.g., one with 4 stations or 10 stations, or one that has stations S3a/b/c rather than just S3a/b? Explain your approach.
- How easy will it be to alter your program to cater for a production line that is more complicated than the "straight line" widget processing that your current program does, e.g., one that involves taking two different types of widgets and assembling them to make a new type of widget in Fig. 2? Would you design your program differently if you had known that this might be a possibility? Explain your approach.
- Conclusion

6 Marking Criteria

You will be graded according to the following marking criteria:

Program Correctness

- Production Line Design and Implementation **[35%]**
- Discrete Event Simulation Design and Implementation **[20%]**
- Calculations and Output Generation **[20%]**

Report [15%]

Miscellaneous

- Code Format & Comments **[10%]**

Other Deductions

- Errors with input, filenames, submission, execution, and mathematical results will attract penalties.

Note:

- Assignments submitted after the deadline, without approved extension, will be penalized by 10% per day late (including weekend).
- The assignment (code and report) will be checked for plagiarism. A plagiarized assignment will receive a zero score and be reported to SACO.
- As per the Student Conduct Rule , any work submitted for assessment must be your own original work, and as such, the use of AI LLMs such as ChatGPT and other similar tools cannot be used in the writing or drafting of any work (which includes both code and report) submitted for assessment. By breaching this rule, you will be at risk of being reported to SACO and receive a zero score.

7 Hint

This section includes some of the important hints mentioned in class and answers some of the queries that were received.

7.1 Simulation Logic

Note that there are different possible solutions to this problem. They will be acceptable as long as you have demonstrated proper use of OO design with respect to inheritance and polymorphism. **Please note that the usage of concurrency (semaphores, threads, etc.) is not allowed**. You may refer to the following points for a basic working implementation which also helps with unstarving and unblocking (this is a general guidance and you are free to come up with creative solutions):

- A priority queue can be set up to manage all the jobs in the simulation.
- Jobs can be thought of as the tasks of pulling a widget into a stage or pushing a widget out of a stage. This jobs will be labelled by a location (next stage/queue) and time (latest time spent in stage/queue).
- At the start of your program, you may loop through all the stages and each stage will attempt to pull a widget into the stage. This means the program is constantly checking for unstarving. However, proper care needs to be addressed for the beginning stages as we will be creating widgets and not pulling widgets.
- Whenever a stage is being blocked, the current job that could not be pushed can be temporarily stored in a separate list. The stage then continues pulling to process new widgets.
- After a widget is processed, you may reintroduce the blocked jobs from the temporary list into the priority. Remember to update the simulation time on the job before reintroduction. This process will then go on until a stage is unblocked.
- The simulation must halt at 10,000,000 time units and any ongoing processes must be terminated.

7.2 Output Check

You may take note of the following to make sure that your implementation is correct:

- The starve time for stages `S0a` and `S0b` will always be zero.
- The block time for stage `S6` will always be zero.
- The number of widgets through each possible production paths should be fairly even.
- Given test values of $M = 1000$, $N = 100$ and $Qmax = 10$, the total number of widgets at the end of line will be approximately 10000, with more originating from `S0b` compared to `S0a`.
- During debugging (not final submission), you can obtain a repeatable set of pseudo-random numbers by fixing the seed value of your random number generator. Assuming the seed value is set at 100, you will have:

```
Random r = new Random(100);
```