SENG2200 Programming Languages & Paradigms
School of Electrical Engineering and Computing
Semester 1, 2019

Assignment 3 (100 marks, 15%) – Due Tuesday June 4, 23:59

1. Objectives

This assignment aims to do practices on OO design and Java programming in topics of *interface, polymorphism, generics* and *iterator*. A successful completion should be able to demonstrate a solution of assignment tasks with correct Java implementation and report.

2. Classes and Responsibilities

You will note that, for this assignment, you are not being given much of an indication of what the classes should be, where to use Inheritance and Polymorphism, and how to allocate responsibilities to your classes. You will need to spend some time thinking about how to design this program, and then doing an actual design. There is at least one clean and efficient design for this program, and if you finish up having difficulties in working out which object should be doing a particular task, then it is possible that you need to go back to your design and check over its clarity and simplicity.

3. Discrete Event Simulation

You will use Discrete Event Simulation for this assignment. While the assignment at first glance may look like an application for concurrent programming, you are NOT to use it.

3.1 A Production Line

For this assignment you will write a Java program to simulate the production of "items" on a production line. The production line will consist 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 an item 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.

Production at any stage for this program will consist of

- taking an item from the preceding inter-stage storage;
- calculating how long it will take to process through this stage (using a random number generator);

- then after that amount of time, placing the item into the following inter-stage storage.

For this simulation, production times are calculated according to a uniform probability distribution. This is easily catered by using the standard Java random number generator from **java.util**. After setting up a random number generator with

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

you can obtain the next pseudo-random number (in the range 0 to 1) from **r** using

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

Given mean M and range N, the time will simply be **P=M+N*(d-0.5)**. If the current time is T1, then you will know that this stage completes production on this item at time

```
T2 = T1 + P.
```

So far, so good, except that the production stages by having varying production times, and only finite sized storages between them, will affect the ability of those stations either side of them to produce their next item.

3.2 Blocking and Starving

If a stage finishes processing an item but finds that its destination storage is full, then that stage must ***block*** until its successor stage takes an item from the storage, thereby allowing it to re-submit its item to the storage and continue with its next item.

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

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

3.3 Unblocking and Unstarving

When a stage completes production on an item, it must check whether stages **either** side of it, which are currently blocked or starved, may now be able to resume production. For example, if a stage is blocked it is because the storage following it is full, so if the stage after it starts production on a new item, it must have taken that item from the storage between these two stages, so there is now a free space for the earlier stage to place its item into the storage and so it can do so, and go ahead and try to begin production on its next item. Similarly, if a stage is starving and the stage before it has just completed processing an item, then the later stage can resume production by taking that item from the storage between the two stages.

3.4 Time

Time will be simulation time, as calculated above and will step forward according to which production stage finishes its current item next.

This will therefore be a simple discrete event simulation, which can be controlled by time events that are placed into a *priority queue*. At any time, this queue will have a maximum length equal to the number of production stages, and so this queue may be implemented using a simple list with "insert in sorted order, remove from front" (there is no need for a heap-based priority queue).

Time will start at zero and proceed until the time limit of the simulation. Time is probably best stored as a double, that way the chance of two time values being equal to within 13 significant figures is so remote that it may be ignored.

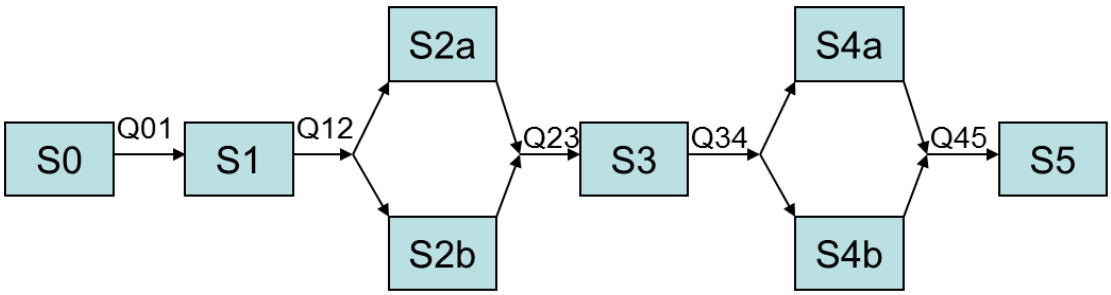
Note: You are allowed to use standard Java container, such as PriorityQueue and ArrayList.

3.5 Items

An item could be anything from cars to mousetraps. What is important in this simulation is the amount of idle time at each station, how long it takes a normal item to be produced, and what is the overall throughput of the production line.

For this assignment an item 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, so that we can calculate things like the average time waiting in each queue, etc.

3.6 Our Production Line



It can be seen from the diagram that S0 is the beginning stage, and S5 is the final stage. Stages S2a and S2b, and S4a and S4b are parallel stages and share entry and exit storages.

As an initial experiment, M and N will be 1000 time units for all stages except S2a/b and S4a/b, for which M and N will both be 2000 time units (to keep the line balanced). The

inter-stage storage capacities will initially be set to 5. The simulation time limit will be 10,000,000 time units, so the production line will produce about 10000 items. You should simulate using a range of values of N (within physical limits – no negative times) and inter-stage storage values between 3 and 7 to see how the efficiency of the line varies.

4. Input and Output

1. The values of *M*, *N*, and *Qmax* are to be read in from command line, in form of (using example values):

   ```
   java PA3 1000 1000 7
   ```

2. All results are to be produced on standard output.
3. The program should produce statistics on the amount of time each stage spends in
   a. actual production (as a percentage)
   b. how much time is spent starving
   c. how much time is spent blocked.
4. For the inter-stage storages, calculate
   a. the average time an item spends in each queue
   b. the average number of items in the queue at any time (this last statistic will require some thought).
5. Lastly you will also
   a. keep a total of the number of items created by **each path** (for example, *S0* → *S1* → *S2b* → *S3* →*S4a* → *S5*) that arrive at the end of the line.

5. Hints

- Begin by implementing stages S0, S1, S2a, S3, S4a, and S5 as a straight through balanced line (M=N=1000), and then add in S2b and S4b (altering other stations or storages as necessary).
- Spend time on designing your suite of classes, a good design will more than halve your workload. Design the stages and storages first. Look for similarities between the different stages and storages to decide how inheritance and polymorphism can be best used in your design.
- You can obtain a repeatable set of pseudo-random numbers from `r` by providing a particular seed value as a parameter to the construction of `r`, eg

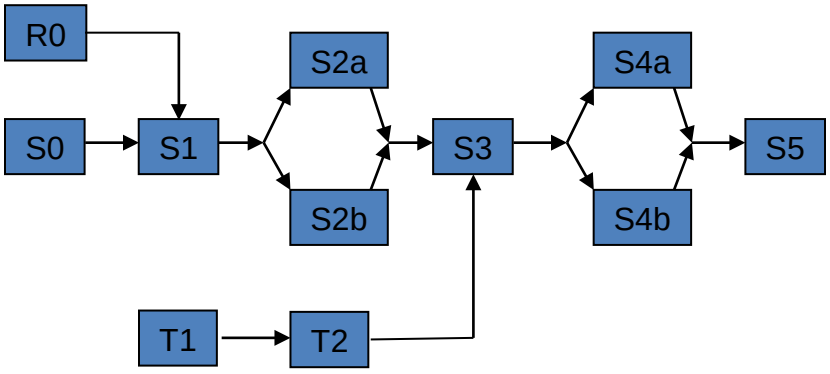
```
Random r = new Random(7.89);
```

This allows you to run your simulation many times with the same sequence of random numbers by way of the r.nextDouble() method call given above.

6. The Written Report

(You will probably need several pages to do this properly – you will also need a significant amount of time to think through and answer the later questions properly.)

1. Keep and present records of how much time you spent on designing, reviewing, coding, testing, and correcting (within these various phases of) your assignment development.
2. Produce a UML class diagram that shows the classes (and interfaces) in your program and the relationship(s) between them.
3. Comment on your use of Inheritance and Polymorphism and how you arrived at the particular Inheritance/Polymorphic relationships you used in your program.
4. 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 2 a/b/c rather than just 2 a/b?
5. How easy will it be to alter your program to cater for a production line that is more complicated than the "straight line" item processing that your program does – e.g. one that involves taking two different types of items and assembling them to make a new type of item? Would you design your program differently if you had known that this might be a possibility? E.g. the following production line?



7. Submission

Submission is through the Assessment tab for SENG2200 on Blackboard under the entry Submit PA3. If you submit more than once then only the latest will be graded. Every submission should be ONE ZIP file named c9999999PA3.zip (where 9…9 is your student number) containing:

- Assessment item cover sheet.
- Report (PDF file): addresses the tasks of Section 6, the written report.
- Program source files.

7.1 Coding Language and Compilation

Programming is in **Java**, and will be compiled against **Java 1.8**, as per the standard university lab environment. Name your startup class **`PA3`** (capital-P capital-A number-3), that is the marker will expect to be able compile your program with the command:

```
javac PA3.java
```

and to run your program with the command:

```
java PA3 1000 1000 7
```

within a **Command Prompt window**, where the integers are *M*, *N* and Q_{max} respectively.

It is expected that **`PA3`** will a) be in the *root of your submission folder*, and b) *compile the entire project* without any special requirements (including additional software). You may include a **`readme.txt`** file, if you require any special switches or compilation method to be used; but this may incur a penalty if you stray too far from the above guidelines.

8. Marking Criteria

*** If your submission cannot be compiled or has run-time errors (e.g., segmentation fault), you may receive ZERO mark.*

*** The mark for an assessment item submitted after the designated time on the due date, without an approved extension of time, will be reduced by 10% of the possible maximum mark for that assessment item for each day or part day that the assessment item is late. Note: this applies equally to week and weekend days.*

This guide is to provide an overview of Assignment 3 marking criteria. It briefly shows marks of each section. This guide is subject to be adjusted.

A detailed deduction checking list will be available on Blackboard.

Program Correctness (85%)

- ***Production Line and Objects*** **(40%)**
  - *Incl. Stage and InterstageQueue class*
  - *Incl. Design and OO-Principles*
- ***Simulation and Statistics Gathering*** **(25%)**
  - *Incl. Item and Event*
  - *Incl. Time Management*
  - *Incl. Block and Starve Management*
- ***Output Form*** **(20%)**

- *How the output is presented, including alignment and clarity of results.*
- *The correctness of calculation.*

Report (15%)

- Clarity, correctness, content and formatting.

Other Deductions (UP TO 30%)

Some other possible deductions may be applied to your submission. For example (not limited to):

- *Incorrect output results, missing or out of range output.*
- *Late submission (this will NOT be capped by 30% of total marks)*
- *Coversheet*
- *Other*