The University of Newcastle
School of Electrical Engineering and Computing
SENG2200 Programming Languages & Paradigms
Semester 1, 2018
Programming Assignment 3 - Due Sunday June 3, 23:59pm

Production Line

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.

Required Solution

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 Q_{max}). The inter-stage storages are necessary because the time taken to process an item at any production stage will vary due to random factors.

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) and 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 for by using the standard Java random number generator from `java.util`.

After setting up the 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.

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 stages (except the beginning stages) in a starved state, and all the inter-stage storages empty.

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.

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 priority 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"*.

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.

Items

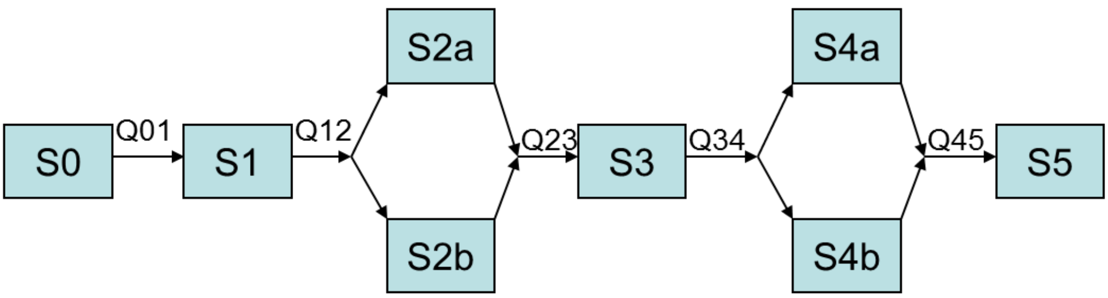
What they are does is of no consequence. They can be anything from cars to mousetraps. What is important in this simulation is the amount of idle time at each station, how long it takes a normal item to be produced, and what is the overall throughput of the production line.

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.

Discrete Event Simulation

You will use Discrete Event Simulation for this assignment. This will be explained in class if necessary. While the assignment at first glance may look like an application for concurrent programming, **you are NOT to use it**.

Our Production Line



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

NOTE: items created in *S0* are free to travel through ANY stage once on the production line; they are **NOT** restricted having to pass through the other *Sxa* stages. For example, an item may take the path *S0* → *S1* → *S2b* → *S3* →*S4a* → *S5*. In other words when an item in queued before a parallel stage, the item (regardless of its origin) will be passed to the first available stage.

The simulation time limit will be ***10,000,000*** time units.

Stages *S2a and S4b* will have *mean M* and *range N* values of **`2 * M`** and **`2 * N`**, while stages *S2b* and S4a will have *mean M* and *range N* values of **`M`** and **0.5** * **`N`**. Other stages are will have mean *M* and range *N* values of simply **M** and **N**. The *inter-stage storage* size Q_{max}, will be always be `> 1`.

You should test your simulation using a range of *range* values of *N* (within physical limits – no negative times) and *inter-stage storage sizes* of Q_{max} between *3* and *7* to see how the efficiency of the line varies.

Data Structures:

Having now *'rolled your own'* data structures, you may now make use of either your own existing data structure from earlier assignments (with appropriate additions or modification), or make use of the Java Collections Framework (see: https://en.wikipedia.org/wiki/Java_collections_framework).

Input and Output:

The values of *M*, *N*, and Q_{max} are to be read in from the command line, in the form (*using example values*):

`java PA3 1000 1000 7`

All results are to be produced on standard output.

1. 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.
2. 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*).
3. Lastly you will also 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.

Hints:

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);
```

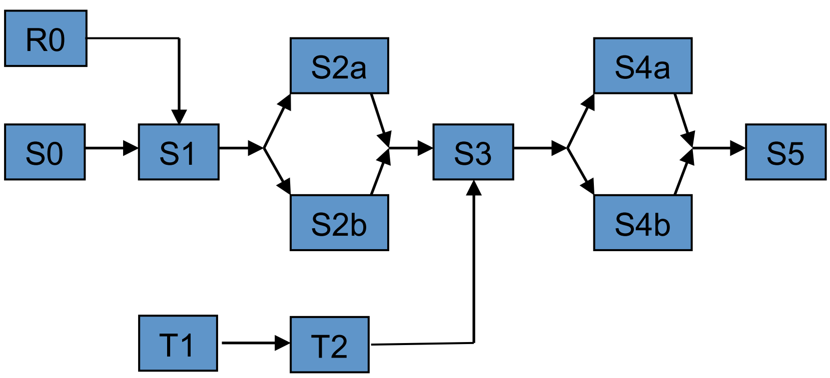
This allows you to run your simulation many times with the same sequence of random numbers by way of the **r.nextDouble()** method call given above.

The Written Report:

Your Written Report is worth 20% of the assignment's value, and will address each of the following:

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 3 a/b/c rather than just 3 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?



(*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*.)

Submission:

Submission is to the ***Assignments Tab*** for SENG2200 on Blackboard, under the entry **A3 Submission**. This allows you to submit the assignment for it to be available for grading.

If you find errors after submission, then you may re-submit your assignment, but only the last submission will be graded. The written report is to be submitted as part of the online submission outlined below, and so you need to finish the program with enough time to spare in order to complete the written report.

Remember to include an *Assessment Item Cover Sheet* with your submission.

You are to write your program in Java, with the driver classes called **PA3**. Please submit a single file called **`c9999999PA3.zip`** (where 9…9 is your student number); do not submit **`.rar`** or **`.gz`** or any other type of archive file. Your submission must contain a) *an assessment item cover sheet*, b) the *written report*, and c) *your program source files*.

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.