Discrete Event Simulation Report

Sean Crocker c3307768

Assignment Development

Designing

The assignment began by reading through the specification and determining which objects, data structures and/or collections were needed to suit the requirements. Initially, the specification looked like it was asking to develop a linked list with stages and storages only to later realise that the two types would cause type errors when using generics. The storage would be created using a queue data structure and the stages would be an abstract class with the Parallel Stages, Standard Stages, Beginning Stage, and Final Stage being concrete classes. An object would be created called item that would have a time and an ID. An object called time would also be created storing the waiting time and leaving time. After time, the realization that time must be stored in the object was clear and that to find the average time and items for the queue a class for statistics may need to be created or the statistics would be stored in the queue. A driver class called PA3 must be created and must accepts arguments to set the mean, range and maximum queue length. A class for the production line must also be created, storing the queues and the stages. The starved and blocked time for each stage must be stored in the stage.

Reviewing

After determining that a linked list would not work, the production line had to be handled another way. After reviewing that the time class would be unnecessary, designing the project without it was easy. There was lots of confusion throughout the project determining how to link the queues and the stages and if a collection of queues and stages were necessary. A lot of time was spent reviewing whether another queue class should be created to store the queues, whether the stages were meant to be a part of the queues or vice versa and whether it only needed one queue for the entire simulation. A large amount of reviewing was also done considering how to initiate the simulation. After reviewing, realization that the starved and blocked time for each stage must either be stored in a class for statistics or in the stage. After reviewing the total blocked time displaying zero the realization that the stages were never blocked due to the queues never being full. These queues were never full due to the incorrect use of the simulation, the items were going through the stages one-by-one instead of determining which stage had the least amount of time. This resulted in a redesign of the production line. The priority queue had to remove values in order to retrieve them, therefore a new queue or array list must be created to transfer these values into. This way of doing things seems wrong and unnecessary.

Coding

The coding began by creating all the necessary classes and work through them one at a time. The PA3 class first, implementing the method to read the arguments and store them. After that, storing the arguments in the production line had to be done, so implementation on the production line class was done. After this, the storage queue was implemented, incorporating basic queue functions and making sure the size was set with the value given. A custom queue interface was created after this. After the storage, a basic stage was created that later turned into an abstract class for other stages. The standard stages, first stage and last stage were created from this with the intention to add parallel stages after the simulation is working. Stages' next and previous storage locations were added. In the driver class, each individual Stage and storage was created making the code look messy and undesirable. This

would later be fixed using a loop. Due to multiple stages sharing the same methods, an interface was created for the stages. The item class was created, and generics were then added to each class to accept items. Various toString methods were added to the stages, storages and the production line to test the functionality. IDs and names were added to storages and stages to present the results as well as make it easier to link the queues with the stages. After the realization that the production line was not accepting values properly, the production line method for beginning the simulation was changed to determine which stage had the lowest production line. This was done by implementing a priority queue and a comparator followed by an array list of all the stages.

Testing

After testing the simulation for the first time, null point exceptions and out of bound exceptions would occur due to logical mistakes. After fixing these, the results for the time starved and time blocked would return 0. As well as the average time and average items. After reviewing my code and design I was able to fix the total starving time but not the blocking time. The average time for the queue were displaying negative results. After redesigning the reviewing the issue was resolved. After implementing the fix for the blocked time, the times were still displaying zero.

Correcting

The null point exception was fixed by reviewing the code and realizing that the items were never created. The out of bounds exception was fixed by fixing a for loop when implementing data for the queues. The starve time was fixed by redesigning the way the production line entered data and how the stages totaled the starved time. The average time was fixed by redesigning the item class so that incorporates an array of doubles for each queue so that it may set the entered and leaved time for a queue. The queue would also store every item that is dequeued in an array list so that the average time could be calculated by getting the leave time and taking away the entry time for all items and adding them together. This result would be divided by the array list’s size.

Class Diagram

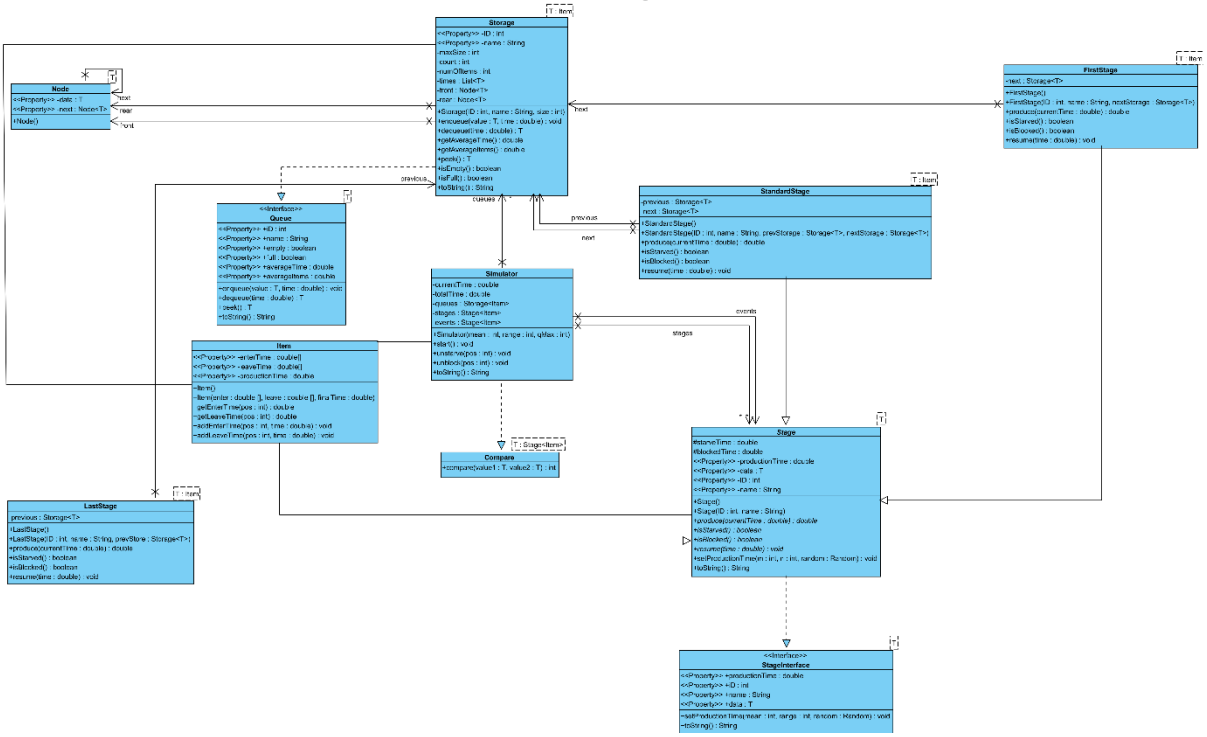

Figure 1 Class Diagram

Polymorphism and Inheritance

Polymorphism

The use of polymorphism was used by incorporating an abstract stage class and extending different types of stages from it. The standard stage, beginning stage, final stage, and parallel stages were all stages must incorporate the methods differently. Determining whether they were blocked or starved were done differently and determining how to produce an item was different. Using this was determined by the way different stages handled data.

Inheritance

The use of inheritance was used by extending interfaces to classes such as the production line, the storage and the stages.

Different Topology or Production Line

Altering the program to include different amounts and types of stages would be too complex as there currently is no way to determine how many stages there are. However, adding the additional parallel stages would not be too hard.

Altering the program to include a different type of production line would not be easy, the whole program may have to be redesigned as the program only caters for one type of item. Only one item is allowed to be stored in a stage at any given time.