SENG2200 REPORT

PA1 Program Analysis

Contents

APRIL 21, 2017

BENJAMIN HOGAN
c3256846

Student Name: Benjamin Hogan
Student No: c3256846

Introduction

This report displays the number and type of errors throughout the project's design and coding stage. For PA3 of SENG2200 Paradigms and Languages, errors were mainly found in methods which involved looping, arrays and formulas. The report clearly states where the errors were found and how they were corrected. A UML diagram has been included and other questions related to the program are below.

Design and Development

My design of PA3 involves 8 classes. The main is contained in PA3 which simply creates a Simulation object. The Simulation class does most of the heavy lifting and is where the main loop of the discrete event simulation runs. Our production line consists of Stations and Queues. Every Queue has a max size determined by the user and is consisted of nodes of type Object in a Linked List Structure.

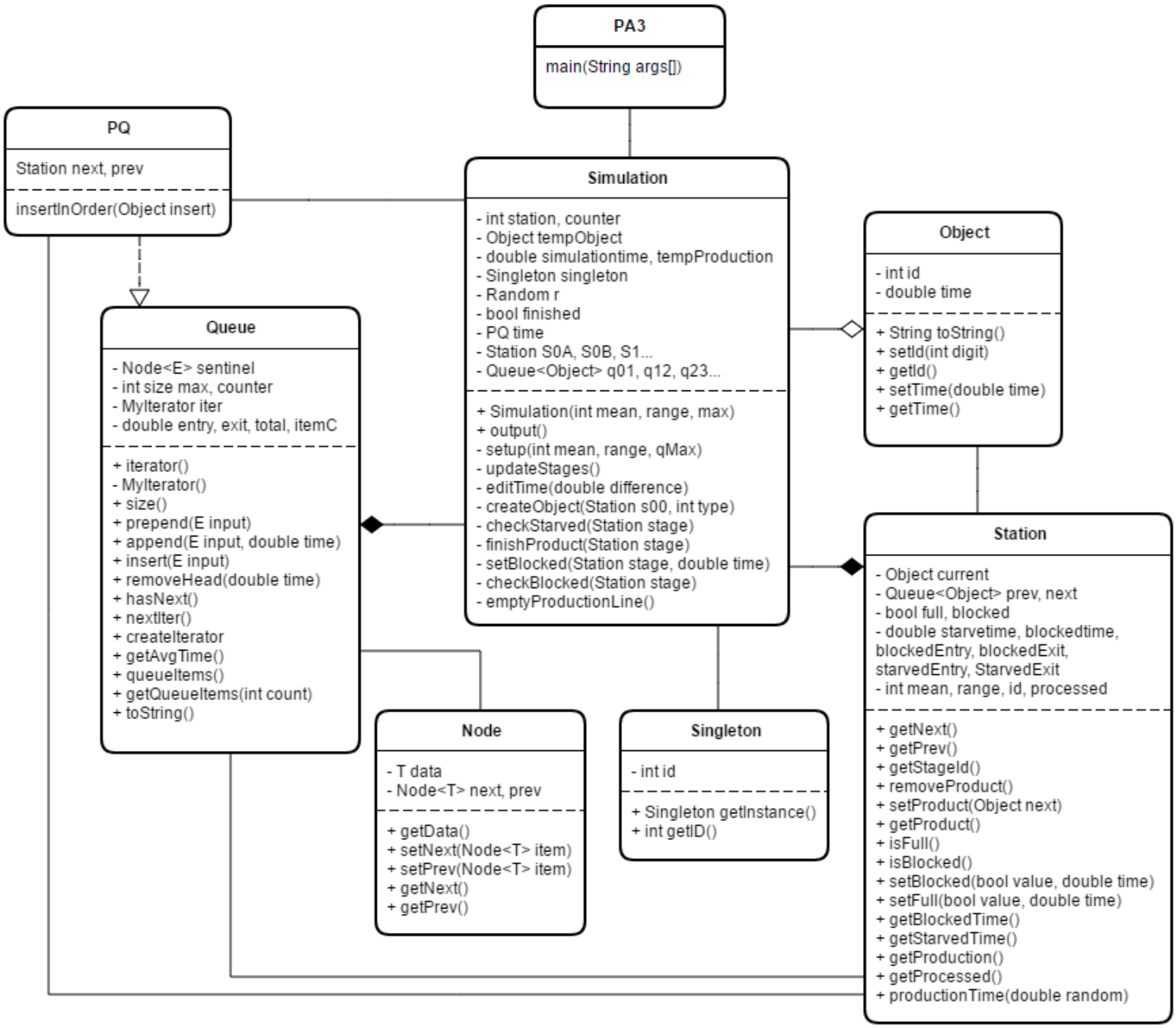
The Station class can hold one Object at a time and has a next and prev Queue to pass on finished objects and start processing a new object. The PQ class is an extension of Queue and is used purely for time priority queueing. The time PQ runs the main flow of the program by removing the head of the PQ and also removing time from the simulation and the other time intervals already in the priority queue. Objects are created in S0A and S0B and are given a unique id from the singleton class. Once the simulation time as exceeded the limit, emptyProductionLine() is called which releases all objects stuck in the line at the end of time. Then all output is displayed in a table with queue and station statistics.

1. Time spent on Classes:

Class	Time Spent	Amount of Errors
Object	30m	0
Singleton	10m	0
Station	1h	2
Node	5m	0
Queue	1h	1
PQ	30m	0
Simulation	10h	10
PA3	5m	0
Commenting and styling	1h	-
Total:	14h 20m	13

Student Name: Benjamin Hogan
Student No: c3256846

2. UML Diagram



Student Name: Benjamin Hogan
Student No: c3256846

3. Inheritance and Polymorphism

From the UML diagram above, the PQ class is an extension on Queue. PQ gains all functionality of a Queue object, hence the relationship between them. PQ has extra functionality by adding an insertInOrder() method which sorts objects in their time order, allowing the updateStages() method to remove the head node in the PQ as it has the highest priority. Therefore, through polymorphism, the compiler can determine whether the object is a Queue or Priority Queue at run time.

4. Extra functionality

Altering the linear production line structure to cater for a different topology would be quite simple. For a topology with more or less stations, it is easy to implement in my program. Having a s0a, b, c structure simply would require a new station, and linking between the queues before it. This will cause more blocking issues but it is easily implemented in my program. Below is an example of adding more functionality to the program:

- Adding a new if statement. For the a, b, c structure it would simply have 4 cases instead of 2 for checking starved/blocking shown below S8a, b, c.

```
if(station == 9) // S9
{
    try{
        finishProduct(s9);
    }catch(NoSuchElementException e) // catches the exception
    {
        setBlocked(s9, simulationTime);
    }
    // checks stations either side for blocked or starved
    checkStarved(s8A);
    checkStarved(s8B);
    checkStarved(s8C);
    checkBlocked(s10);
}
```

5. Complex Production Line

Similarly, to the extra functionality question, to have a program which creates 2 different objects and requires two different objects to create a new one would need a slightly different design for each station to handle different types of objects. It would require a new station and/or queue for each new stage, then the same process by checking behind and in front for starving or blocking. Then passing the object onto the next queue. Then for the different object, simply starve until 2 objects are in the station, then finish production on that object by creating a new one. The process is listed below:

Add the new queues and stations.

```
private void setup(int mean, int range, int qMax)
{
    q01 = new Queue<Object>(qMax);
    q12 = new Queue<Object>(qMax);
    q23 = new Queue<Object>(qMax);
    q34 = new Queue<Object>(qMax);
    q45 = new Queue<Object>(qMax);
    q56 = new Queue<Object>(qMax);
    // NEW Queues with max size

    s0A = new Station(null, q01, 2*mean, 2*range, 0);
    s0B = new Station(null, q01, mean, range, 1);
    s1 = new Station(q01, q12, mean, range, 2);
    s2 = new Station(q12, q23, mean, range, 3);
    s3A = new Station(q23, q34, 2*mean, 2*range, 4);
    s3B = new Station(q23, q34, 2*mean, 2*range, 5);
    s4 = new Station(q34, q45, mean, range, 6);
    s5A = new Station(q45, q56, 2*mean, 2*range, 7);
    s5B = new Station(q45, q56, 2*mean, 2*range, 8);
    s6 = new Station(q56, null, mean, range, 9);
    // NEW STATIONS with next and prev queues.

    // create objects in stations for updateStages to start.
    createObject(s0A, 0);
    createObject(s0B, 1);
    // any additional new creation stages for the start process
}
```

Then update the stages by calling updateStages(), then checking either stages beside it.

```
if(station == 9) // S9
{
    try{
        finishProduct(s9);
    }catch(NoSuchElementException e) // catches the exception and sets
    the station as blocked
    {
        setBlocked(s9, simulationTime);
    }
    // checks stations either side for blocked or starved
    checkStarved(s8);
    checkBlocked(s10);
}
```