SENG2200 PA3 Report

Jordan Haigh - C3256730

Journal Entries

Monday 8 May 2017 – 3 Hours

I have started reading through the assignment specification and making notes of what I need to do to complete this assignment. With the assistance of two students, we have discussed how to go about completing the assignment.

Dan has created a Discussion Board post about patterns.[1] The post goes on to talk about the Singleton Pattern[2] and how it would it beneficial to this assignment. Whilst reading through the websites that have been provided, they started to go into more detail about using multithreading and synchronization. Dan has clarified that the Singleton Pattern will be used in its simplest form. The pattern has been added in the form of a class, ensuring that only one instance of a class is created.

Most of the work today has been understanding how to approach the assignment.

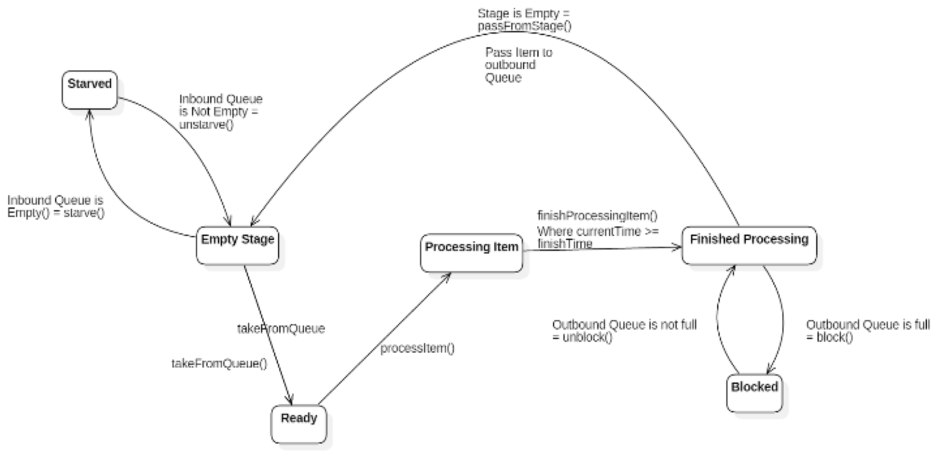
Friday 12.5.17 – 1.5 Hours

I have started work on the UML diagram before diving head first into coding. The design will be a crucial part of this assignment, as I tend to be a visual learner. If I can get the diagram looking like it will work, I can make a start on the code. Using StarUML[3] will be useful as it can generate Java code from the methods and attributes specified in the diagram.

Saturday 13.5.17 6 hours

I completed a UML design that I am happy with now. This design is using an abstract Stage class that extends out to a 'StartStage' and 'EndStage' class. It also includes a MasterStage class which holds a list of Stages, used when parallel Stages are required. StarUML has generated the code for all classes and I have started working from the ground up. The Singleton implementation was gathered from one of the links that Dan provided on the Discussion Board. The Item class has also been implemented, using a UniqueID generator from the Singleton Class. I have created unit tests for these classes to make sure that their functionality is bulletproof.

More class methods have been implemented, including the Stages and the start of the Simulation class. I have decided to implement a State Enumeration to determine which state the Stage will be in. Using a UML Diagram has assisted in understanding the how each state will work with another.


Stage is Empty = passFromStage()
Pass Item to outbound Queue
Starved
Inbound Queue is Not Empty = unstarve()
Inbound Queue is Empty() = starve()
Empty Stage
Processing Item
finishProcessingItem() Where currentTime >= finishTime
Finished Processing
takeFromQueue
takeFromQueue()
processItem()
Outbound Queue is not full = unblock()
Outbound Queue is full = block()
Blocked
Ready

[1] https://www.tutorialspoint.com/design_pattern/design_pattern_tutorial.pdf

[2] http://www.javaworld.com/article/2073352/core-java/simply-singleton.html?page=1

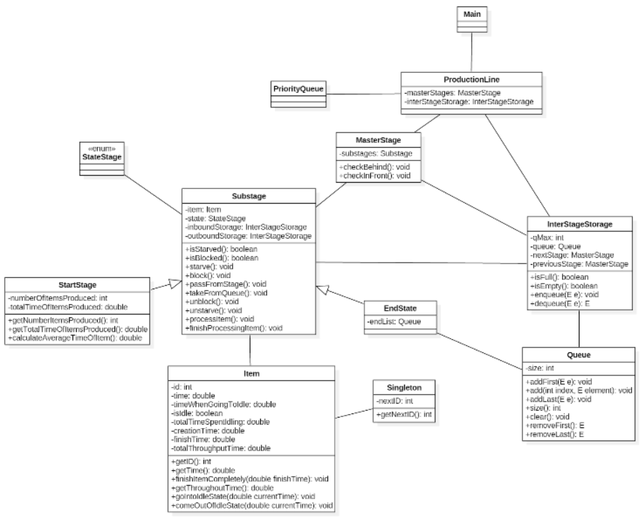
[3] http://staruml.io/

Sunday 14.5.17 - 7 hours

I have spent extra time creating unit tests making sure that other classes are performing the correct functionality.

Most classes have enough functionality to start stepping through the production line and seeing if they can process items. I have decided to work with the Observer Pattern to pass the random ‘p’ value calculated in the SubStage class, back up to the ProductionLine Class.[4]

At this stage, the UML diagram is starting to get messy, with many classes communicating with each other using tight coupling. This led to a redesign of the program with a different approach.

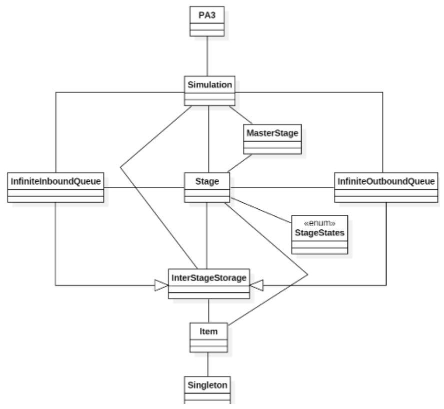


Old design of the Production Line

I have decided to start small, using a linear simulation without the hassle of parallel stages. If I can get a single stage simulation working, I can later expand out to use parallels.

The new diagram has removed the MasterStage class for the meantime, as well as StartStage and EndStage. It has been replaced with an InfiniteInboundQueue class and InfiniteOutboundQueue class that extend from InterStageStorage.

The current design is much easier to follow and seems more logical using an Infinite Inbound/Outbound to handle to generation and storing of items.



New design of the Production Line

Monday 15.5.17 - 5 hours

I have started coding from the ground up again, some classes remaining the same as before(Item and Singleton). The Stage class is still using the StageState Enumeration to track what state the Stage is in during the Simulation.

The InfiniteInboundQueue and InfiniteOutboundQueue extends from the InterStageStorage and overrides some of the existing methods. I have included two observer subjects:

- InfiniteInboundQueue observing Simulation – Monitoring the current time from the simulation and creating new items with that current time
- Simulation observing Stage – Monitoring the P Value calculation to determine when a stage will finish processing an item.

[4] https://en.wikipedia.org/wiki/Observer_pattern

At this point in time, the UML diagram is starting to look messy again with multiple observer patterns, especially with the Simulation being both a Subject and an Observer.

Everything seems to be working properly up to this point in time, I have yet to run the simulation to determine whether Items can be passed through stages to queues, and queues to stages.

Tuesday 16.5.17 - 2 hours

I have come to the realization that the Observer Pattern isn't the best tool in this situation. The Stage class now receives a Simulation parameter in its constructor, making the Observer pattern redundant for this assignment.

I have started stepping through the simulation to see if everything is working correctly. The first stage can receive an item from the Infinite Inbound Storage and process for X amount of time. Once the current time matches its finish time, it successfully sends the item to the outbound storage.

Saturday 20.5.17 - 7 hours

Most of the time spent today was dedicated to getting the simulation working using two stages and an intermediate queue. If I can get this small section working, then I can expand it out to more stages and queues. Using a whiteboard to visualize items (using magnets) moving through the production line was very helpful as I tend to be a visual learner. This also helped in determining 'if condition' checks for the simulation.

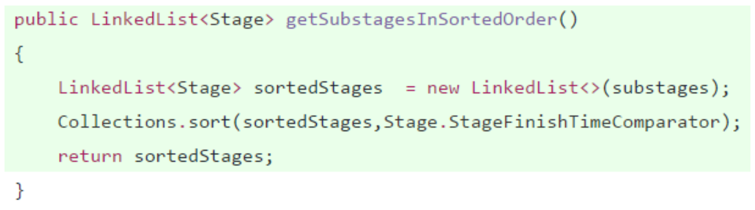
The Stage class needed a small amount of redesign due to Chaining Function Calls inside one another, when they were not supposed to. Extra accessor methods were added to determine what stage the stage was in.

The Simulation has started working correctly using the modified production line. For the moment, I have edited the constructors (multipliers set to exponential integers) and P Calculation (doesn't calculate according to the specification and simply returns the multiplier) so that I can track the simulation without using doubles. This is purely for debugging right now and will revert to the original before submission.

Now that a linear simulation is working correctly, I can now implement the Parallel stages. This is through the MasterStage class that contains a LinkedList of Stages. The MasterStage will include similar functionality to the Stage class (Forward and Backward MasterStages, Inbound and Outbound Storages). To establish a link between each stage to go inside a MasterStage, the Stage class now includes a setParent() method.

The MasterStage implementation wasn't as difficult as expected (another for-each loop). The only problem that I face now is that since the loop goes through the list of stages, it places a priority on the first stage in a parallel set. So that the first stage will be unblocked, even if the second stage has been blocked for a longer period of time.

Sunday 21.5.17 - 7 hours



```
public LinkedList<Stage> getSubstagesInSortedOrder()
{
    LinkedList<Stage> sortedStages  = new LinkedList<>(substages);
    Collections.sort(sortedStages,Stage.StageFinishTimeComparator);
    return sortedStages;
}
```

To overcome the stage priority issue, I have created a comparator method that can be passed through to a linked list constructor. The comparator will sort the Stages inside the linked list by the finish time values. Using the Collection.Sort() method, it will sort a copy of the existing LinkedList.

With this fix implemented, I can now start creating the production line as specified in the assignment. I am still using fixed numbers for each stage, only to make certain that the simulation is handling the unblocking correctly.

The simulation is running correctly, now I can revert to the correct formula for calculating the finish time and make a start on the data statistics.

Most of the data statistics will be easy to implement, for example when determining processing, blocking and starving, whenever an stage's state is updated, a tally will be incremented.

Calculating the Queue statistics is taking more time to complete. The average time in a queue had to be calculated using three linked lists (LinkedList for time when enqueuing, LinkedList for time when dequeuing and LinkedList for the time difference for each element). This isn't the best approach to solve this issue, but for the meantime, it is working.

The average number of items in a queue at a given time will take a lot more thought.

Monday 22.5.17 – 3 Hours

This morning I have been working on finding an expression that will determine the number of items in the queue at any given time. This has involved a lot of whiteboard work to make sure that the expression makes sense and works for the simulation. In a simple simulation (S0→ QueueSize2 →S1) to determine the average time of items in a queue with a MAX_SIMULATION_TIME of 500. The end calculation determined that there were 1.994 items in the queue at any given time, which seemed to be a valid result.

The command line arguments have been implemented so that the user can input their mean, range and qMax.

Thursday 25.5.17 – 2 Hours

With the help of a peer, I was able to fix my Queue calculation involving three very long LinkedLists. It has also come to light that I was incorrectly using the random number generator and that the program doesn't work with seeds.

I need to fix this by putting the Random number Generator outside of the stage class and in the main class, so that the entire program can work with the seed. I have tried to implement this, but to no avail. My program is incorrectly outputting statistics which are illogical.

Friday 26.5.17 – 10 Minutes

The Random Number Generator issue was due to passing the same double through to the P value calculation rather than the Random object itself. This issue has been fixed and the program now works with seeds.

Sunday 28.5.17 – 1 Hour

After reviewing the assignment specification for final touches, I have found a section that requires clarification. The specification states to store data statistics inside the item class for the time when the item is created, finished and idle etc. My Stage and Queue class handles all the relevant statistics for the output. I cannot understand how to implement the Item statistics so that it may be used in the Stage and Queue class. I have emailed Dan regarding this for explanation.

Monday 29.5.17 – 1 Hour

Until I hear back from Dan, I will add the implementation for relevant methods and update their statistics, though they will not be printed in the output (uses null checks to eliminate NullPointerExceptions The output specification only asks for stage and queue statistics, which have been implemented respectively.

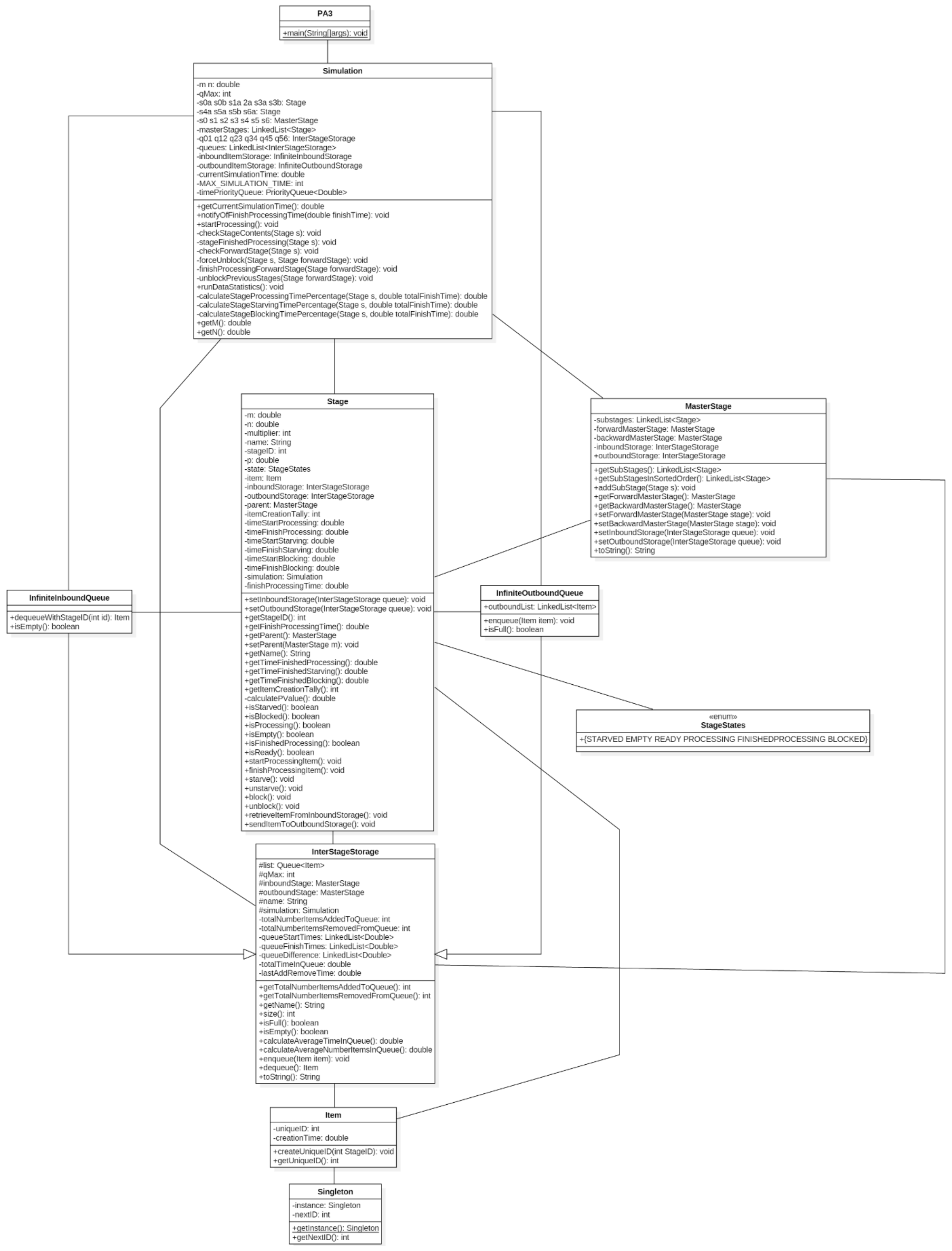
Time spent Designing, Reviewing, Coding, Testing and Correcting

Throughout this assignment, I went through four different program designs before finding a design that I was happy to use. These designs were slowly built upon one another. The time periods above determine how long the assignment took to complete. There were multiple times where I was unhappy with the code I had written and found a simpler solution to the problem.

I believe the main issue I had was diving head first into the assignment using parallels right from the start. This caused mass confusion and several headaches to wrap my head around. By reverting to a simple linear simulation, it made the concept of a production line much easier to understand. Expanding upon the production line later was simpler once the ground work had been created. A majority of testing was done by stepping through the code line by line and using a whiteboard with magnets to track each item throughout the simulation.

With the coding side of the assignment, I had to restart only once with a completely new design. There were a few times where exceptions were being thrown, but that was only in the event of missing precondition checks.

UML Diagram


PA3
+main(String[]args): void
Simulation
-m n: double
-qMax: int
-s0a s0b s1a 2a s3a s3b: Stage
-s4a s5a s5b s6a: Stage
-s0 s1 s2 s3 s4 s5 s6: MasterStage
-masterStages: LinkedList<Stage>
-q01 q12 q23 q34 q45 q56: InterStageStorage
-queues: LinkedList<InterStageStorage>
-inboundItemStorage: InfiniteInboundStorage
-outboundItemStorage: InfiniteOutboundStorage
-currentSimulationTime: double
-MAX_SIMULATION_TIME: int
-timePriorityQueue: PriorityQueue<Double>
+getCurrentSimulationTime(): double
+notifyOfFinishProcessingTime(double finishTime): void
+startProcessing(): void
-checkStageContents(Stage s): void
-stageFinishedProcessing(Stage s): void
-checkForwardStage(Stage s): void
-forceUnblock(Stage s, Stage forwardStage): void
-finishProcessingForwardStage(Stage forwardStage): void
-unblockPreviousStages(Stage forwardStage): void
+runDataStatistics(): void
-calculateStageProcessingTimePercentage(Stage s, double totalFinishTime): double
-calculateStageStarvingTimePercentage(Stage s, double totalFinishTime): double
-calculateStageBlockingTimePercentage(Stage s, double totalFinishTime): double
+getM(): double
+getN(): double
Stage
-m: double
-n: double
-multiplier: int
-name: String
-stageID: int
-p: double
-state: StageStates
-item: Item
-inboundStorage: InterStageStorage
-outboundStorage: InterStageStorage
-parent: MasterStage
-itemCreationTally: int
-timeStartProcessing: double
-timeFinishProcessing: double
-timeStartStarving: double
-timeFinishStarving: double
-timeStartBlocking: double
-timeFinishBlocking: double
-simulation: Simulation
-finishProcessingTime: double
+setInboundStorage(InterStageStorage queue): void
+setOutboundStorage(InterStageStorage queue): void
+getStageID(): int
+getFinishProcessingTime(): double
+getParent(): MasterStage
+setParent(MasterStage m): void
+getName(): String
+getTimeFinishedProcessing(): double
+getTimeFinishedStarving(): double
+getTimeFinishedBlocking(): double
+getItemCreationTally(): int
-calculatePValue(): double
+isStarved(): boolean
+isBlocked(): boolean
+isProcessing(): boolean
+isEmpty(): boolean
+isFinishedProcessing(): boolean
+isReady(): boolean
+startProcessingItem(): void
+finishProcessingItem(): void
+starve(): void
+unstarve(): void
+block(): void
+unblock(): void
+retrieveItemFromInboundStorage(): void
+sendItemToOutboundStorage(): void
MasterStage
-substages: LinkedList<Stage>
-forwardMasterStage: MasterStage
-backwardMasterStage: MasterStage
-inboundStorage: InterStageStorage
+outboundStorage: InterStageStorage
+getSubStages(): LinkedList<Stage>
+getSubStagesInSortedOrder(): LinkedList<Stage>
+addSubStage(Stage s): void
+getForwardMasterStage(): MasterStage
+getBackwardMasterStage(): MasterStage
+setForwardMasterStage(MasterStage stage): void
+setBackwardMasterStage(MasterStage stage): void
+setInboundStorage(InterStageStorage queue): void
+setOutboundStorage(InterStageStorage queue): void
+toString(): String
InfiniteInboundQueue
+dequeueWithStageID(int id): Item
+isEmpty(): boolean
InfiniteOutboundQueue
+outboundList: LinkedList<Item>
+enqueue(Item item): void
+isFull(): boolean
«enum»
StageStates
+{STARVED EMPTY READY PROCESSING FINISHEDPROCESSING BLOCKED}
InterStageStorage
#list: Queue<Item>
#qMax: int
#inboundStage: MasterStage
#outboundStage: MasterStage
#name: String
#simulation: Simulation
-totalNumberItemsAddedToQueue: int
-totalNumberItemsRemovedFromQueue: int
-queueStartTimes: LinkedList<Double>
-queueFinishTimes: LinkedList<Double>
-queueDifference: LinkedList<Double>
-totalTimeInQueue: double
-lastAddRemoveTime: double
+getTotalNumberItemsAddedToQueue(): int
+getTotalNumberItemsRemovedFromQueue(): int
+getName(): String
+size(): int
+isFull(): boolean
+isEmpty(): boolean
+calculateAverageTimeInQueue(): double
+calculateAverageNumberItemsInQueue(): double
+enqueue(Item item): void
+dequeue(): Item
+toString(): String
Item
-uniqueID: int
-creationTime: double
+createUniqueID(int StageID): void
+getUniqueID(): int
Singleton
-instance: Singleton
-nextID: int
+getInstance(): Singleton
+getNextID(): int

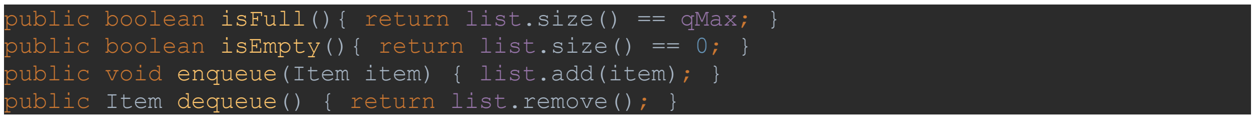
Inheritance and Polymorphism

My original design was going to use a concrete *Stage* class, and extend that class to a *StartStage* and *EndStage* class in the program. This would allow for the starting stage to take in an *InterStageStorage* that would always create items, similarly, the ending stage would use an *IntersStageStorage* that would always hold items.

I wasn't happy with this design when it came to the coding side of the assignment, and decided to use a different approach to the assignment.

My next and current approach was to create a concrete *InterStageStorage* class and extend that to an InfiniteInboundStorage class and an *InfiniteOutboundStorage* class. This would allow for a simpler design, whilst sticking to my original design.

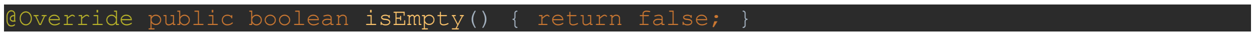
InterStageStorage includes methods for determining whether the queue is empty or full, enqueuing and dequeuing. The implementations for these methods include:



```
public boolean isFull(){ return list.size() == qMax; }
public boolean isEmpty(){ return list.size() == 0; }
public void enqueue(Item item) { list.add(item); }
public Item dequeue() { return list.remove(); }
```

Note that the enqueue and dequeue methods are missing their statistical implementation in the image, this is only for explanation of Inheritance and Polymorphism in this Assignment.

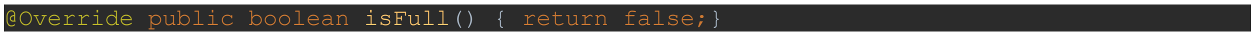
The *InfiniteInboundStorage* Class has been defined to create an infinite number of items (within the max simulation time), so that the forward Stage can never starve. Therefore, it must override the *isEmpty()* method so that this class will never be empty, and the starting stage(s) is always able to collect a new item.



```
@Override public boolean isEmpty() { return false; }
```

The *dequeue()* method in the *InfiniteInboundStorage* is different from the parent's implementation since it must generate a StageID with the Item being generated. To accommodate for this, the Stage class must check whether the *inboundQueue* variable is an instance of *InfiniteInboundStorage,* so that it can call the appropriate method.

The *InfiniteOutboundStorage* class has been defined to hold an infinite number of items, so that the last stage in the production line can never be blocked. To achieve this, the *isFull()* method inherited from the parent class must be overridden so that this class will never be full.



```
@Override public boolean isFull() { return false;}
```

This class implements a *LinkedList<Item>* so that is it able to append an infinite number of items (or until the program crashes from running out of memory).

Most constructors across the program have one or more implementations. The stage class contains two implementations for the constructor, one using the ID for the stage (starting stages utilise this constructor), whereas all other stages use the second constructor implementation.

```
public Stage(int multiplier, int stageID, Simulation simulation, String name)
public Stage(int multiplier, Simulation simulation, String name)
```

Similarly, with the InterStageStorage class, one constructor defines a Queue Max variable, where one does not. This is to accommodate for the child classes of InterStageStorage (InfiniteInboundStorage and InfiniteOutboundStorage) since they do not need a maximum length of their queues.

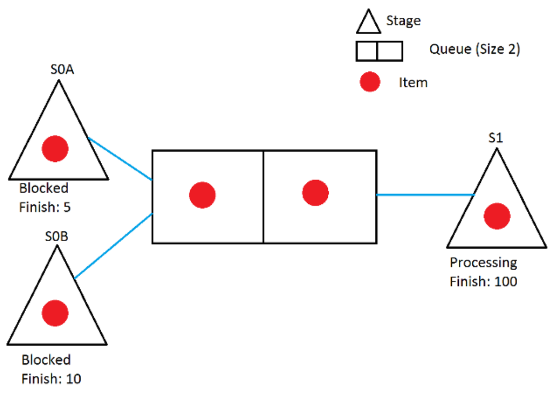
```
public InterStageStorage(MasterStage inboundStage, MasterStage outboundStage,
Simulation simulation, String name)

public InterStageStorage(int qMax, MasterStage inboundStage, MasterStage outbound-
Stage, Simulation simulation, String name)
```

Altering the Production Line for Expansion

My Simulation program has the functionality for a different topology. The MasterStage class handles parallel stages, with the usage of a LinkedList of Stages (allowing for possibly an infinite number of substages in a MasterStage). The length of the production line can be extended by creating new Stages and InterStage storages. Each stage uses a multiplier in its constructor to determine the possible processing time of an item.

The major work will be creating the links between each stage's forward and backward queue, and vice versa. I have created private methods inside the simulation for initialising and linking each element of the simulation so that another developer can understand each part of the setup in the Simulation constructor.

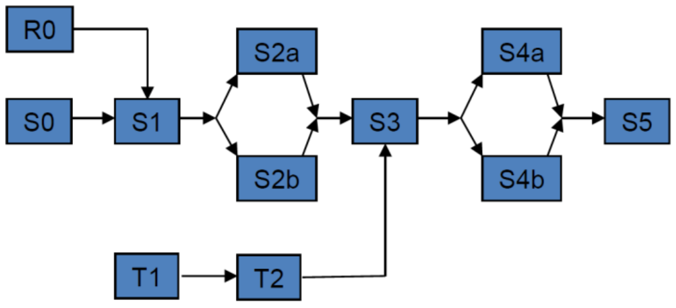
The only concern for expansion of the production will be the time taken to complete the maximum allocated simulation time. My Simulation works through every MasterStage and each Substage (in order of what is meant to finish processing first).



This means that if all current parallel stages are blocked and the forward stage is still processing, all current sub-stages will still perform the check to see if the forward stage has finished (even though it is determined from the first substage that has tried to unblock itself) See the image to the right for this scenario. In the event of a linear system (without parallel stages), this issue would be resolved.

If I had more time to complete the assignment (or redo the assignment again later), I would use a different setup for the Priority Queue, by passing in the Stage and the time the Item will finish processing (achieved using a tuple). This would increase efficiency as each removal from the Priority Queue would move straight to the Stage in question, ready to finish processing. The only concern would then be how to handle unblocking once the Stage has attempted to finish processing.

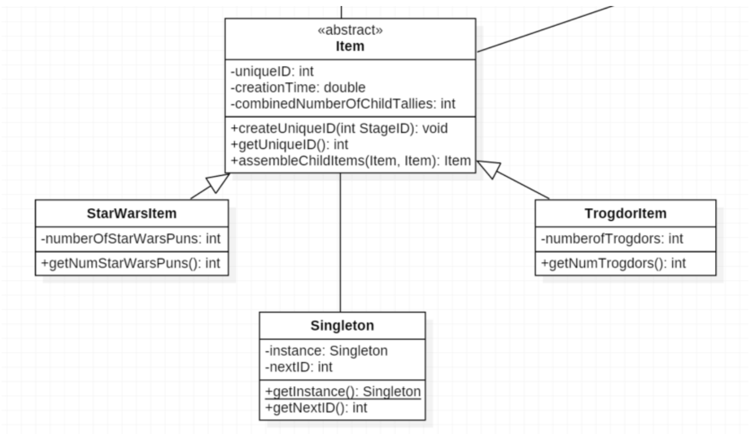
Altering the Production Line for a Complex Item Processing



In the event of expansion to cater for more complex processing, the underlying structure of the Simulation class would remain the same. The Simulation class has the functionality to expand to allow for more parallel stages and/or more MasterStages.

The example given in the assignment specification "One that involves taking two different types of items and assembling them to make a new type of item" may change the underlying structure of the program.

A complex production line example found in the assignment specification (image shown above), includes the MasterStage S3 having multiple backward MasterStages. For the program to satisfy this, the functionality will need to be altered to allow for any number of previous MasterStages (in the form of a LinkedList<MasterStage>). Because of this, the running time of the program will be affected depending on the complexity of the production line



To allow for more than two or more different types of items to exist in the program, the Item class could be declared abstract (with the same implementation) so that child classes of the Item class can exist. The overall program would still work with an Item class, though now because of abstraction, we could use the child classes of the Item class instead.

The child classes of the Item class can contain their own statistics; for a simple example, they may be integer tallies.

The now *Abstract Item* class can introduce a method (assembleChildItems(Item, Item): Item) to combine the two unique variables (numberOfStarWarsPuns and numberOfTrogdors) into a single variable (combinedNumberOfChildTallies) and continue to work with a combined Item object.

This could be extended to implement a Production Line of Production Lines, so that each component is created, processed and assembled in the parent production line. A good example of this is a car production line. Each individual part of a car is created and processed separately in its own production line. The car can then be fully assembled when all parts have been created and processed.