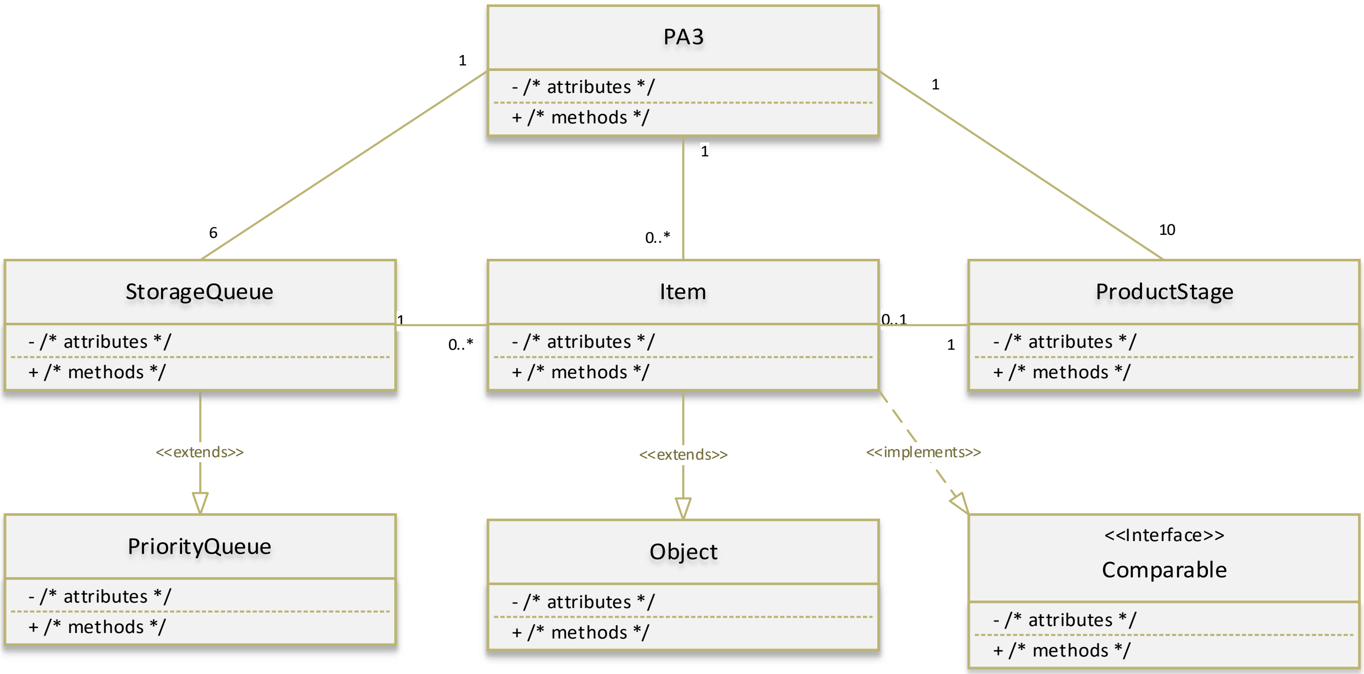
SENG2200
ASSIGNMENT 3 REPORT

1. UML diagram for Assignment 3:



2. The use of Inheritance/Polymorphism

Inheritance played a crucial role in this assessment, despite the relatively small number of classes used. The StorageQueue class extended the standard Java PriorityQueue, and the Item class extended the generic Object class while implementing the Comparable interface, greatly reducing the overall workload. These two classes allow the developer to very easily create, manipulate and queue a newly-defined object.

In the final product the StorageQueue class is extremely small – just one attribute and one method were added. The only thing that the super class PriorityQueue lacked was a name, which was important especially for the output.
This deviated from the initial outline where the StorageQueue class was used for Starve and Block checks as well. It quickly became apparent that those checks were better used in the ProductStage class.

I made the decision to extend Object for the Item class in the initial outline. Object has a lot of useful generic functionality that allows it to synergise well with the PriorityQueue class, and the StorageQueue class by extension. Like the StorageQueue class, the Item class adds attributes and methods needed for the assignment output – the more practical functionality is handled by the Object class.

The Item class did not implement the Comparable interface in the initial outline. An unforeseen side-effect of extending PriorityQueue and Object appeared to be that the interface was mandatory to prevent compiler-time errors, despite adding no comparing functionality. This effectively means that, although the Item class implements the interface, Comparable goes unused in the program.

3. Updating functionality for different production line topology

Several changes need to be made to the program to allow for a different path. Most significantly, the stages and queues are hardcoded. This is unavoidable – the layout of the stages and queues does not follow any obvious pattern and attempting to define the production line procedurally does not add any more flexibility. In the program the queues are defined in class PA3, lines 16-18, and the stages are defined in class PA3, lines 15 & 21-46. These need to be updated to change the production line.

Next is the unique IDs. Currently there are 2 variables that control IDs: idNumberA and idNumberB, found in class PA3, lines 50-51. idNumberA creates unique IDs for items created in stage 0a and idNumberB for those made in stage 0b. This is fine for any program that has 2 potential starting stages, but suppose it had 50. An array of unique ID seeds would be much more useful in that situation. On the other hand, idNumberB is useless for a production line with only 1 starting stage.

Only a few small changes to the conditionals in the do-while loop in class PA3, lines 53-146 are really necessary. Besides that, the StarveTimeByStage method in class PA3 only allows for production lines with splits of up to 2 stages. This could be easily updated by using a local integer to track how many stages are in a "group".

There are solutions to most of these problems that can be solved in a few lines of code each. The only problem with no simple procedural solution is the first one – unless the stages of the production line follow a predictable pattern (e.g. 0a/0b -> 1 -> 2a/2b -> … -> na/nb), they must be hardcoded with each change.
One point not raised in the first paragraph is that, since the inter-storage queues link one "group" of stages to another (for example, queue Q01 might link 0a/0b/0c to 1a/1b/1c/1d), queues can be defined procedurally, unlike stages.

4. Updating functionality for more complex production lines, beyond the "straight-line" used in the program

The program would not be designed significantly differently if these changes were made. Each item has a full record of the stages it went through from start to finish. Suppose the end product is a plane if the item starts at R0 and a car if it starts at T1. This information is in the attribute *stages*, making it simple to express through discrete-event simulation.

A few questions arise though. Will all completed items go to the same storage or is there a separate storage for each type of item? For the former, there are no changes. If the latter, how many possible different items can be created? Using the example production line, there are likely 3 different types of item, 1 for each origin point (R0, S0, T1). Although this can be hard-coded with an array of 3 ArrayLists, a better solution would be an ArrayList of ArrayLists – check the prefix of the *stages* attribute for items as they are completed, create a new ArrayList whenever an item with a different origin is detected and push the item to the appropriate ArrayList based on the prefix. Without additional information, this assumes a lot about how the production line is structured and what the assessed requirements of the program are.
The main change that will need to be made to the program seems small but is deceptively problematic – each stage may have more than 1 previous inter-storage queue. A major part of the Starving functionality is checking the size of the previous queue. In the current program this is simple; there is only 1 previous queue so only that 1 queue needs to be checked. Applying FCFS would be best, to give all potential previous queues a fair share.

The *prevQueue* attribute in the ProductStage class would need to be changed to an array or ListArray, as well as a method and attributes used to determine the next queue.