Johan Bjerkem and Tinus F Alsos

Assignment 3: Wafer Production Line

Testing:
In this assignment we have taken our testing to the next level. We have started using the package *unittest*, which is a part of the Python standard library. This means our testing follows a 'best practice' of writing tests. We have also started using *mocks*, *setUp* and *tearDown* for making our test classes independent and easier to read. Look inside the folder tests to have a look.

Task 1:
A class is created for each object type and is found in the corresponding .py file. As an example, the Buffer class is found in buffer.py. We have made it such that each class prints necessary information whenever there is a change of state.

-Batch (in batch.py):

Has a size and an id number. Batch objects are sent around in the production line, but this class is not "aware" of this. There is also a generator function in this file that generates new batches and numbering them. This is useful for printing out what batches are being processed where in the production line at a given time.

-Buffer (in buffers.py):

Has a capacity, id number, and some identification of special buffers (first buffer or last buffer). The primary functionality of the buffer is that you can reserve space, load a batch to the buffer and unload a batch from the buffer. The buffer manages a list of batches that are in the buffer at any time. Buffers are loaded and unloaded by the FIFO principle (First In, First Out).

The first buffer is special because we have set its capacity to infinite. The reason for this is that this allows us to flood the system, and rather let the system choose intelligently what to process when. In theory, having all batches in the initial buffer will never be worse than loading batches into the initial buffer at predetermined times, since the system can also be adjusted such that the batches in the inputbuffer are also processed at predetermined times.

-Task (in task.py):

Has a task number and input- and output buffers to know where to load batches from and where to load batches to. The most important functionality of the task is to process a new batch. By processing a new batch, we mean to unload a batch from the inputbuffer, **calculate the time it takes to process the batch**, then load the batch to the output buffer after the time it took to process the batch has elapsed.

-Unit (in unit.py):

Has a unit number and a list of tasks that are part of the unit. It also has an import variable idle which is a boolean saying whether the unit is idle (and therefore may process a new batch in one of its tasks) or if it is not idle (meaning that one of its tasks is currently processing a batch). The most important function is to process a new batch, which starts the processing of a batch in one of its tasks if it has a batch available.

-ProductionLine (in productionLine.py):

Is a class that connects the different parts of the production line together. All it does is create instances of batches, buffers, tasks, and units according to the description of the production line. The only other functionality it has is to load a batch to the production line by adding an input batch to the first buffer.

Task 2 and 3:

We have chosen to implement a discrete event simulation. The simulator is managed through an event manager that manages events, and a simulation that engages the event loop until the simulation is finished. All the above classes from the production line (except for the batch class), takes as input an EventQueue instance as well as a time such that events are triggered at correct times and triggered events cause new events to be added to the simulation.

-EventQueue (in EventManager.py):

The event queue uses events to ensure that everything in the production line happens in chronological order. It can take a new event and add it into the queue, which is always ordered by time. The EventQueue updates the time discreetly by iterating through the event queue.

-Event (in EventManager.py):

An event has a function, an object on which the function shall be executed, and a time at which the function shall be executed at. Additionally, it takes in a variable amount of arguments needed to execute the function.

- Simulator (in Simulator.py):

The simulator initiates a production line with our given parameters, loads it with some batches and then runs a simulation. The simulator can also collect all the prints of the elementary classes and create a text file. This functionality makes it easy for us to log our experiments, and later extract the information we are interested in.

Task 4:
We have documented our findings in the folder task_4. Below is a table presenting our most important findings.

Number of batches (with 50 wafers)	Total time
1 batch (50 wafers)	653 minutes
5 batches (250 wafers)	1870 minutes
20 batches (1000 wafers)	6322 minutes

Table 1: Initial simulation results with 1, 5 and 20 batches of size 20.

Task 5:
We have documented our findings in the folder task_5. Below is a table and a graph presenting our most important findings. As presented, a load time of 250 minutes gave the best result.
We also experimented with load times as low as 0 minutes. This would in theory overload the first buffer and result in batches being discarded. We imagine a work around where you have a buffer before the initial buffer which feeds batches as soon as one exits the buffer. Experimenting with different batch sizes, best load times varied, but generally 0 minutes was among the best. We have therefore chosen to use 0 minutes in load time as our general option in the later experiments.

Load time difference	Total time
250 minutes	6161 minutes
0 minutes	6322 minutes
150 minutes	6322 minutes
450 minutes	9290 minutes
653 minutes	13060 minutes

Table 2: Simulation results of 20 batches og size 50 loaded at 'load time difference' into the system.

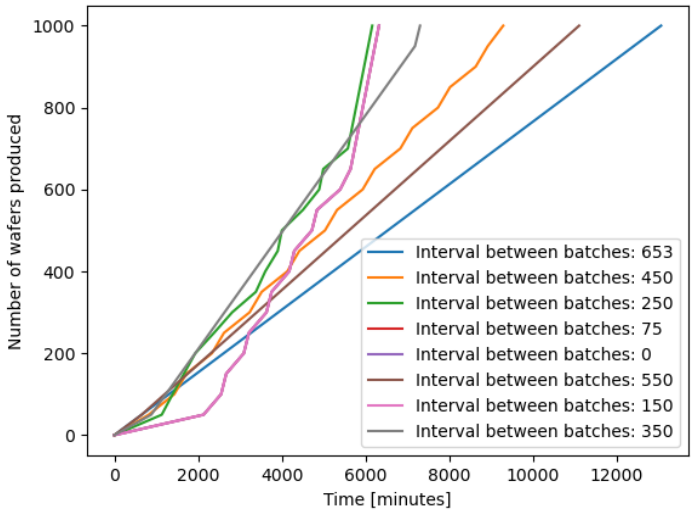

Figure 1: Plot of simulations of 20 batches og size 50 loaded at 'interval between batches' into the system. 'Number of wafers produced' (y-axis) shows the total number of wafers in the final buffer at any time.

Task 6:

We have documented our findings in the folder task_6. We found that the only important heuristic was for unit 2, where task 2 had to be the last priority. This means that a third of all possible permutations of heuristics is optimal.

With the optimal ordering heuristic we got the time 5743 minutes for 20 batches with 50 wafers, and 5724 minutes for 50 batches with 20 wafers. Below is the plot for 50 batches with 20 wafers. The lines ending with shortest time have task 2 as last priority.

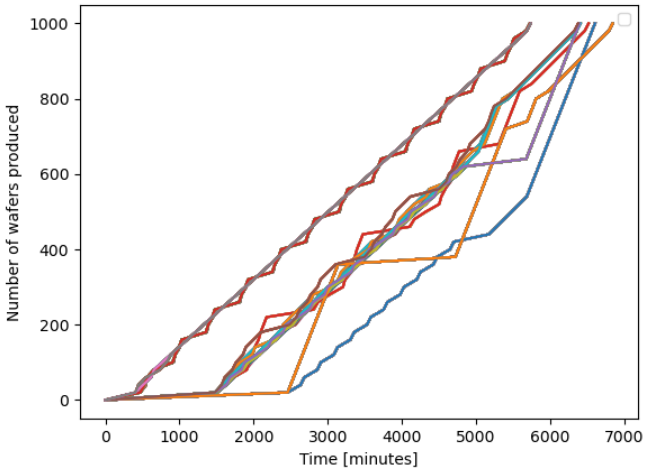

Figure 2: Plot of simulations with different ordering heuristics.

Task 7:
We have documented our findings in the folders task_7. If the 1000 wafers isn't divisible by the batch size we add the rest to the last batch, such that we don't end up with a batch with less than 20 wafers. There doesn't seem to be any underlying pattern to what batch size that's the best. This resulted in our best production time of 5664.7 minutes. The parameters for this result is shown in table 3.

Optimal configuration	
Interval between loading	0 minutes
Ordering heuristic	Unit 1: Tasks 1, 3, 6, 9 Unit 2: Tasks 7, 5, 2 Unit 3: Tasks 4, 8
Batch size	37 (26 batches of size 37 and one batch of size 38
Results 1000 wafers produced	5664.7 minutes

Table 3: Final results and optimal configuration.

Last comment:
In utils.py we have created several functions that help us extract the information we are interested in from the text files created by the simulator. This includes creating plots of the simulations, which you have seen in this report, and extracting the parameters and the times of the best simulations. If you wish to test our simulator, feel free to run main() or main2() in simulator.py to simulate different batch sizes and different ordering heuristics respectively.