Task 1:
In this task we created 6 files; Batch.py, Buffer.py, Task.py, Machine.py, Plant.py and Printer.py, each consisting of a class. In each class we created functions to get, set, delete.etc.

The Batch class manages batches, each with a given batch code, the number of wafers it contains and a state.

The Buffer class manages buffers, each with a given capacity(given constant), total number of wafer, a FIFO-queue of batches, a “historyQueueOfBatches”(history of all batches that has passed through the buffer), source task and a target task. It contains a function to reset itself “resetBuffer”. The “hasSpace” function checks whether it has enough available space to hold an incoming batch. The “enqueueBuffer” function uses the “hasSpace” function to check if there is enough space to hold the incoming batch, before adding it into its que. The “dequeueBuffer” function pops the next batch in que.

The task class manages tasks. It holds a task’s; name, type(event/task), state, incoming buffer, outgoing buffer, holding batch(the batch it performs the task on at a given time), load time(given constant), unload time(given constant) and process time(given constant). Also it contains the function “taskCanBePerformed”, which checks if; there is a batch in the incoming buffer, the task is idle(doesn't hold a batch) and if the output buffer has enough space to hold the incoming batch.

The Machine class manages the machines, each with a name, tasks(in dictionary) and a list containing all possible policy-variants for the machine(used in task 3, the variants is all combinations of the tasks in the machine). The function “resetMachine” resets the machine.

The Plant class manages a plant. From the plant you can control all other classes. It holds a name and lists of all its machines, buffers and batches. It also contains a list “allTasksEvents” which holds all the plants tasks and events, such as “start” and “end”. The function “taskServesBatch” will, as long as the machine isn't busy, dequeue the batch from its source buffer and set the task to process the batch.

In the Printer class we created the “exportPlant” function to print various information about the initiated wafer production plant.

Test(in “Tester.py”): To see information about the wafer production plant, run codeline 50-51 in Tester.py. The result will be exported into a .csv file.

Task 2:
In this task we created the three files Event.py, Schedule.py and Simulator.py. An instance of the class Event can be of two types; either ‘BUFFER_TO_TASK’ or ‘TASK_TO_BUFFER’ meaning that an event moves a batch from buffer to task or vice versa. An event holds a batch, task and buffer, the event number and the date it was performed. The Event-class has different ‘standard’ getters and setters to maintain the accessibility.

The class Schedule holds a plant, a list of current schedules, a current date and an event number. The two main functions in Schedule are ‘scheduleBufferToTask’ and ‘scheduleTaskToBuffer’. These two functions use the function ‘scheduleEvent’, which is a universal function that creates an event, sets the provided batch, buffer and task, and appends this event into the ‘currentSchedule’-list. ‘scheduleBufferToTask’ and ‘scheduleTaskToBuffer’ are called when we want to schedule either action, and later execute the action.

The simulator class holds a plant (the plant that it simulates), a ‘executionTime’ dictionary (which holds the time used for all the batches through the plant), a ‘execution’-list (containing the events that has been executed), the list ‘terminationDates’ (that holds all the termination dates, this is used in task 3), and the list ‘bestTermination’ (which holds the best termination, used i task3).

The main function of the simulator-class is the function ‘simulationLoop’. This loop executes the events in the ‘currentSchedule’ list, as long as this list is not empty. Certain executions generate new schedules, and this feeds the simulation. The function ‘executeEvent’ is called in the simulationLoop, and executes the events based on the event-type. So the ‘executeEvent’ either calls the function ‘executeTaskToBuffer’ or ‘executeBufferToTask’. When either of these two functions are called, new events are scheduled.

Test(in “Tester.py”): To test task 2 and run the simulationLoop and print the execution of five batches, run codeline 56-72 in Tester.py. The result will show in the terminal.

Task 3:
In this task we created the file Optimizer.py, consisting of the class “Optimizer” that holds a name and a list of all possible machine-policy-combinations. The class is used to optimize the total time(sum of the time all batches used to be produced). It is based on varying the different machine operation policies, the sizes of the batches(20-50 wafers) and the different schedules for the introduction of batches into the fabrication process.

The “generateBatches” function uses the “newBach” function to generate new batches into a plant's start-buffer. If “totalNumOfWafers % batchSize > 0”, we produce an extra batch to ensure that enough wafers are produced. In this case, the number of wafers produced will be greater than “totalNumOfWafers”. This a necessity, since all batches have to be the same size (we assume). The function ‘initiatePlant’ calls the ‘generateBatches’ and appends all these batches into the startbuffer of the current plant.

The “generateOperationPoliciesForMachines” generates all possible policy-combinations for each maschin. It does so by fetching a machine's list of containing tasks and then using the permutations function to make a new list of all possible permutations of the machine's task list. Then it appends each policy into the 2D-list “allMachinePolicies”. Then it updates the machines “machinePolicies”-list with the new 2D list.

The “generateAllPossiblePolicyCombinations” uses the “generateOperationPoliciesForMachines” to generate all possible policy-combinations for each machine. Then for each machine it fetches all the machines “all policy-combinations”

and appends it into the “threeDimensionalList”. Then the function uses the product-function and iterates through the product of “threeDimensionalList” before appending each “element” into “allPossiblePolicyCombinations”. Here an element is a 2D list looking like this: [[machine 1’s policy list], [machine 2’s policy list], [machine 3’s policy list]]. The optimizer class holds the list of all possible combinations of these elements called “allPossiblePolicyCombinations”

The “runMachinePolicy” function (from the class Plant) checks if the machine isn't busy and for all the tasks in the machine it checks if; the output-buffer has enough space, the input buffer(sourcebuffer) holds at least one batch and whether the task is busy or not. The priority input is a list of the machine's tasks with the most preferred task to run in the start of the list and the least in the end. For each run the machine uses this priority to know which task to run first, given its possible to run the task.

The optimizer uses a method ‘monteCarloSimulation’. This method generates all possible policy combinations and the list containing all the possible batch-sizes. The ‘MonteCarloSimulation’ then runs the simulation function called ‘simulationLoopForOptimizer’ which is similar to the previous function ‘simulationLoop’. It collects all the terminations, for all types of machine-policies and batch-sizes, in addition to finding the optimal termination (in manner of date, batch-size and machine-policy-combination).

To show the result of the ‘monteCarloSimulation’, we created the two functions ‘plotTerminationDates’ and ‘printHTML’. The ‘plotTerminationDates’ plots a histogram of all the termination dates, showing us which dates are occuring. The ‘printHTML’ shows us this plot, and in addition it shows us two tables showing various statistics about all the terminations and the optimal machine-policy-combinations.

Test(in “Tester.py”): To test the optimizer class, run codeline 77-85 in Tester.py. The result will store a file called ‘optimized.html file that you need to open in your internet-browser. The result should contain a histogram and two tables.